Varqa
A Functional Query Language Based on an Algebraic Approach and Conventional Mathematical Notation.

by
Forouzan Golshani.

A dissertation submitted for the degree of
Doctor of Philosophy

Department of Computer Science
University of Warwick
Coventry
U.K.

August 1982.

Dedicated to

my mother and the memory of my father

who made this possible .

TABLE OF CONTENTS

Abstract

We propose a functional query language for databases where both syntax and semantics are based on conventional mathematics.

We argue that database theory should not be separated from other fields of Computer Science, and that database languages should have the same properties as those of other non-procedural languages.

The data are represented in our database as a collection of sets, and the relationships between the data are represented by functions mapping these sets to each other. A database is therefore a many-sorted algebra; i.e. a collection of indexed sets and indexed operations. As in abstract data type specification, we specify the consequences of applying operations to the data without reference to any particular internal structure of the data.

A query is simply an expression which is built up from the symbols in the signature of the algebra and which complies with the formation rules given by the language. The meaning of a query is the value which is assigned to it by the algebra.

There are several ways of extending our language. Two ways are studied here. The first extension is to allow queries in which sets are defined inductively (i.e. recursively). This mechanism is essential for queries dealing with transitive closures over some interrelated objects.

Secondly, since incomplete information is common to many databases, we extend our language to handle partially available data. One main principle guides our extensions: 'whenever information is added to an incomplete database, subsequent answers to queries must not be less informative than previously'.

Finally, we show the correspondence between Varqa and methods used in current database software. A subset of Varqa, including all features whose implementation is not obvious, is mapped to relational algebra thus showing that our language, though it has been designed with no reference to internal structure, is not incompatible with present database software.

Prologue

The author's first contact with a database system

About six years ago (1976) the author joined a large team whose task was to computerise a large stock control and storage operation system. This system held, among others, information on

- over twenty thousand line items,
- flow of items from supplying sources (through the receiving, storage and shipment departments) to the consumer,
- stock level , and the number of items on order,
- delays, denials and other matters of that type.

One of the first tasks of our team was to conduct a survey on the existing (manual) information handling techniques used in warehouses. Here is a summary of the accounts noted in one typical warehouse:
The warehouse supervisor (the person in charge of the warehouse) had spent over twenty years in exactly that warehouse. He knew the place of almost every single item by heart. (There were over 5,000 items in his warehouse.) Although there was a card system as the location reference, the supervisor used it very rarely. There were relatively few errors and the warehouse staff were capable of correcting them. For example, misplaced items were usually detected quickly and were

moved to their correct locations.

However, the tedious part of the operation was for the staff to keep track of the transactions and input/output of the warehouse. There were masses of paper to be sorted and filed every day. A transaction form had to be filed for every movement of an item. There were more staff busy with sorting and filing than were assigned to all other tasks in the warehouse. In later investigations it was discovered that further "paper work, sorting and filing" took place in two other offices (viz. on the other copies of the transaction forms). This seemed extremely wasteful.

Anyway, despite considerable resistance from warehouse staff, the operation was eventually computerised. The advantages of this computerised system over the old manual one were numerous. The more striking ones were a remarkable speeding up of the whole operation, and relief of the warehouse staff from repetitive and altogether avoidable paper work: a central control system supervised the entire operation.

After a short while, however, occurrences of several errors revealed that we had been naive in thinking that the new system was perfect. The unhappy fact was that these errors manifested themselves in a diverse fashion. Some were human errors (e.g. wrong entries to the system, like punching mistakes) and some were completely unknown and mysterious to us. Great efforts were made to rectify the errors; however, unexpected new cases continued to appear.

Later, we all (both warehouse staff and us) grew to accept the problems of the new system as a fact of life. Whenever anybody asked the project manager about the errors he replied : "Well.... What else do you expect? You know what machines are like."

The project manager was not totally wrong and, anyway, at that time it was beyond our means to do much better. Years later, however, the situation has seen very little improvement. Most experienced computer users are not surprised if they find errors for which no reason exists. Indeed, many users, like our former project manager, do not distinguish between the 'software faults' and the machine's faults.

Perhaps a totally different approach can solve (at least part of) the problem.

Mathematical Notation

We discuss here the set-theoretical, logical and metalinguistic notation which will be applied in this thesis. Part of the set-theoretical concepts derive from ideas presented in [HMT-71].

In set theory, the fundamental relationship between objects is that of membership; we use $x \in A$ to express that object x belongs to the set A. $x \notin A$ expresses the opposite. $A \subseteq B$ and $A \subset B$ stand for "A is a subset of B", and "A is a true subset of B" respectively. The union of two sets A and B is written as $A \cup B$, and their intersection is written as $A \cap B$.

Given a set A and a formula P(x) we use $\{ x \mid x \in A \text{ and } P(x) \}_x$ and $\{ x \in A \mid P(x) \}_x$ to denote the set of all objects from A which satisfy the predicate P(x) , i.e. those elements of A for which the predicate P(x) holds. Thus, $y \in \{ x \in A \mid P(x) \}_x$ iff y is an element of A and P(y) is true. Whenever unambiguous we may omit the trailing subscript x.

For any two sets A and B, the cartesian product of A and B, denoted by $A \times B$, is the set of all ordered pairs <a,b> such that $a \in A$ and $b \in B$. A relation between A and B is any subset of the cartesian product of A and B. (We will see a more general definition of relations in the later chapters.) A function f from A to B , written as f : A --> B , is a relation between A and B where no two pairs have equal first and unequal second members. That is, a relation R between A and B is a function iff

$$\langle a,b\rangle \in R \text{ and } \langle a,b'\rangle \in R \text{ implies } b=b' .$$

The set of all first members of the elements of a function f is called the domain of f and is written as Dom(f). Thus, $Dom(f)=\{a \mid \exists\, b: \langle a,b\rangle \in f\}$. Similarly, the range of function f is

$$Rng(f)=\{ b \mid \exists\, a : \langle a,b\rangle \in f \}.$$

Given a function f:A-->B and $A' \subseteq A$, the restriction of f to A' is the set of all pairs from f whose first member belongs to A'. We write f!A' to denote function f restricted to subset A' of its domain.

In this thesis each non-negative integer is regarded as the set of all preceding natural numbers (they are 0,1,2,3,...): number zero is the empty set $\emptyset$, one is the set $\{0\}$, two is $\{1,0\}$ and so on. Thus , the number n is the set $\{0,1,2,....,n-1\}$. We will see that this notation is remarkably helpful in later formalisms. Given a natural number n, $i \in n$ will be frequently used to indicate that i is one of the natural numbers from the set $\{0,1,2,....,n-1\}$.

In this thesis, a sequence is a function whose domain is the set corresponding to a natural number. For example, a sequence s of length n (i.e. a sequence whose domain is n) has an i^{th} element for every $i \in n$, and is represented as $\langle s_0, s_1, ... , s_{n-1} \rangle$. $|s|$ is the length of the sequence s.

For any set A , **P**(A) is the powerset of A which is the set of all subsets of A , i.e. $D \in \mathbf{P}(A)$ iff $D \subseteq A$.

For any two formulae P and Q , we use P & Q , P v Q , P -> Q and $\neg$ P to denote conjunction of P and Q, disjunction of P and Q, P implies Q , and the negation of P respectively. $\forall$ and $\exists$ stand for the universal and the existential quantifiers respectively. We sometimes use F and T in place of the truth values false and true.

Greek letters will be used as metalinguistic variables.

The structure of this thesis

The chapters in this thesis are designated by Roman numerals I, II,... Reference to a whole chapter is therefore given as, e.g. chapter IV. Major divisions of each chapter are called 'sections'. We identify the sections by Roman capital letters A, B, C,...

The outline of the thesis is as follows. Chapter I contains a brief survey in the state of the field of database systems. An analysis of the existing problems which are relevant to query languages is given. The primary aim of this chapter is to give motivation for the design of a new query language and to state its perspectives.

An informal introduction to the major concepts and constructs of the proposed language is given in chapter II. One of the fundamental concepts is that of algebra: a brief introduction to algebra and the algebraic approach is therefore included in this chapter.

In chapter III we formally define what a database is and give precise semantics of a database and of our language. Several queries against two different databases are formulated and their evaluation is discussed in the final sections of chapters II and III.

In chapter IV we extend our language to cater for certain types of queries which cannot be expressed using the tools provided in the preceding chapters. We introduce first the "where" notation, and then allow queries which define sets inductively. This mechanism increases the power of our language considerably.

Chapter V is focused on a problem which is common to most database systems, namely coping with incomplete information. After studying various aspects relating to nonavailability of data we enrich the language to handle such cases. Obviously when the information in the database is not complete, the answer to any query is only an approximation to the true result. The aim is to get the most precise approximation.

In chapter VI a translation of a subset of Varqa into relational algebra is given. The intention is not to suggest a technique for implementation, but to give a correspondence between our language and methods used in current database software.

Finally, the epilogue gives an outlook on areas where further work can be done. The thesis closes with the list of references.

Acknowledgements

In the preparation of this thesis I have enjoyed constant advice and much encouragement from my supervisor Bill Wadge. This is the place to say a special "thank you" to him.

I would like to thank Tom Maibaum for his valuable comments and stimulating suggestions.

Being a member of the Semantics group has provided the ideal environment in which to write this thesis. Thanks are due to the whole group: Tony Faustini , Steve Matthews, Paul Pilgram and Ali Yaghi.

I would also like to thank all other members of the Computer Science Department for having been of assistance in various ways.

Mrs. Shirley Salmon did the beautiful calligraphies. Many thanks are due to her and her family for being so kind to me during my stay in Britain.

To end the prologue, here are some words about the name "Varqa".

It is a Farsi word (with Arabic origin) and means "(male) nightingale". It was the name of two Baha'is who were martyred in nineteenth century Iran for their religious belief . The name Varqa has been chosen to record the fact that the persecution and execution of the Baha'is is continuing in Iran today.

It is pronounced "varko" in the same rhythm as "panda".

Chapter I

Chapter I

A CRITICAL STUDY OF PRESENT DATABASE SYSTEMS

What can be learned from them?

In this thesis we propose a functional query language based on the notation of conventional mathematics. In the specification of this language, we side with the user and direct our attempts towards the design of a simple language which is also practical from the implementors' point of view.

This chapter contains a brief survey of some aspects of database sytems which are relevant to our proposal. The primary purpose of this chapter is to analyse the existing problems, and justify our attempt for designing a new language.

A. The Problem

In many circumstances concerning the management of data, one is confronted with large amounts of interrelated information. How can it be handled systematically?

This basic question spurred research into the field of databases. The usual problem of representation and formulation of a part of the real world by the computer is of central importance. Only the amount of stored information distinguishes databases from ordinary computerized systems.

There are several descriptions of database systems. A widely accepted one is: "A database is a symbolic representation of knowledge about part of the real world" [We-76]. Or, as in [Da-78] : "a database is a collection of stored operational data used by the application system of some particular enterprise". We are not going to point out the subtle differences between these views (and other views) because, at this level of detail, it is not important. A more detailed investigation of databases reveals a number of critical points. The more important issues in designing a database are:

- how can data be organised?
- what is the best way of storing data?
- how can the stored data be accessed?
- is the database a true representation of the real world?

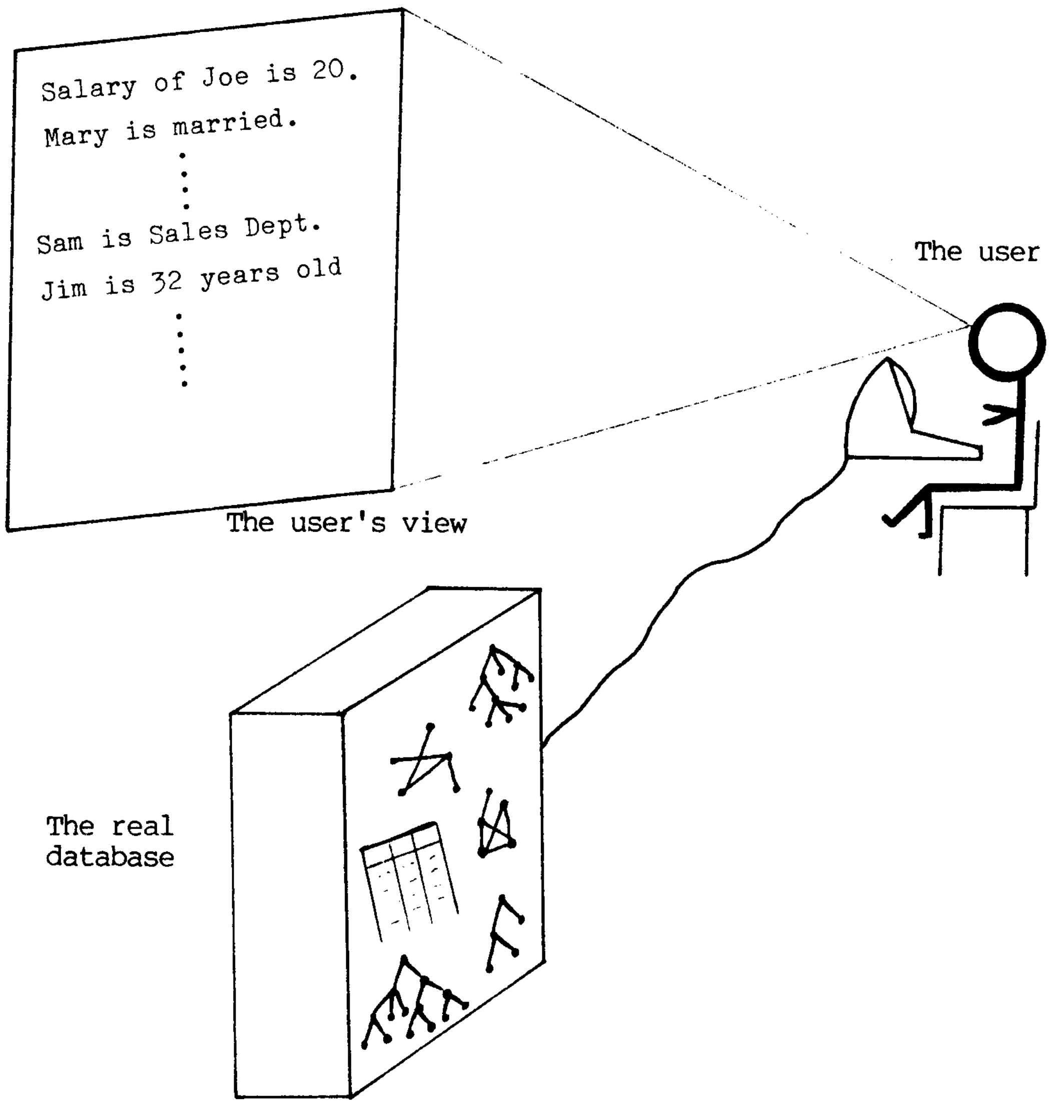
Further questions concern security, maintenance, redundancy and several others.

Each of the above principal issues is rather complex. In the case of accessing the data, for example, the problems range from search strategies within the database files to formulation of various queries. Storage of the data is concerned with the choice of appropriate computer devices, storage architectures and the filing system organisation. In this study, we do not aim at giving solutions to all of these issues. We intend, however, to study and to suggest solutions to some of the problems related to data access by the user.

To explain our goal, we look briefly at the developments in computer languages and, in particular, database query languages. In the early days of computing, the programmer could only be either a computer designer or engineer, or someone with similar knowledge of the system. Programming required specialised knowledge of every detail of the system. Later, with the invention of more suitable hardware and specialised languages, the user only needed to learn the principles of a programming language. However, these principles were, more or less, those of the system itself. In other words, the language designer started from the system designers' view and simplified it for the user. Therefore, the languages were machine oriented. Years after, this method is still in use.

Considering the recent improvements in both hardware and software, we suggest that, in the design of languages, the highest importance should now be given to the users' requirements and not to the hardware and software requirements. This may result in a considerable amount of extra work for the system engineers, but that is what they have been trained for.

Our suggestion is equally important in the construction of database query languages. In the design of our query language, we will side with the user, and further, will keep the user's view clearly separated from the engineer's view of the system.

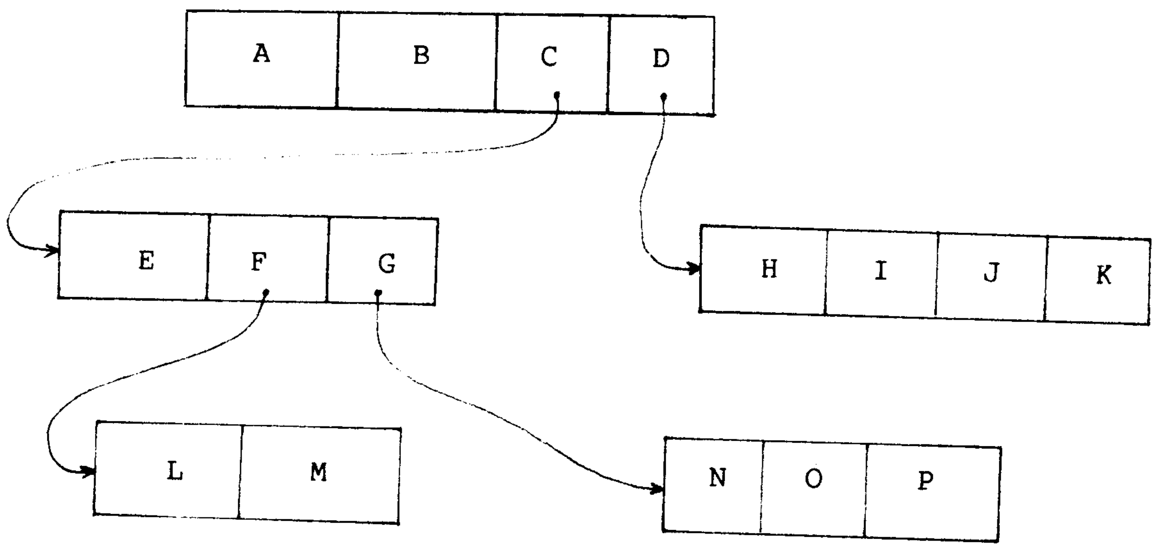


B. Record Based Databases

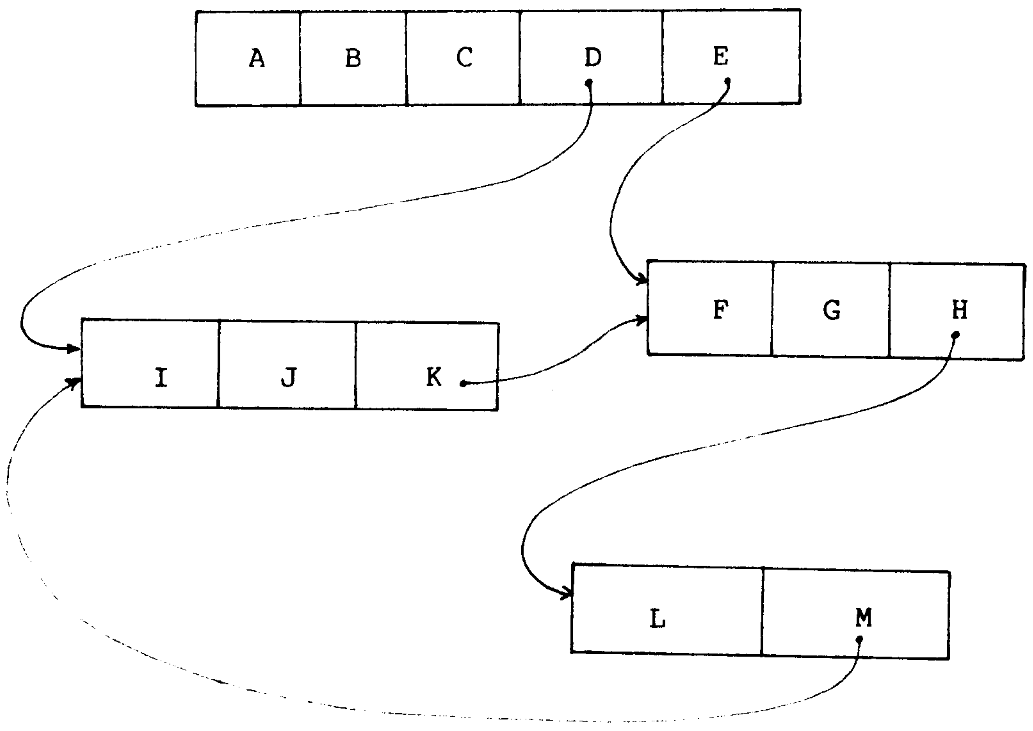
Most database systems fall into three groups: hierarchical, network and relational. Approaches which cannot be categorized as above will be individually studied in the next chapter.

The fundamental concept behind all these models is that of a record. (A record is basically a finite collection of labeled data.)

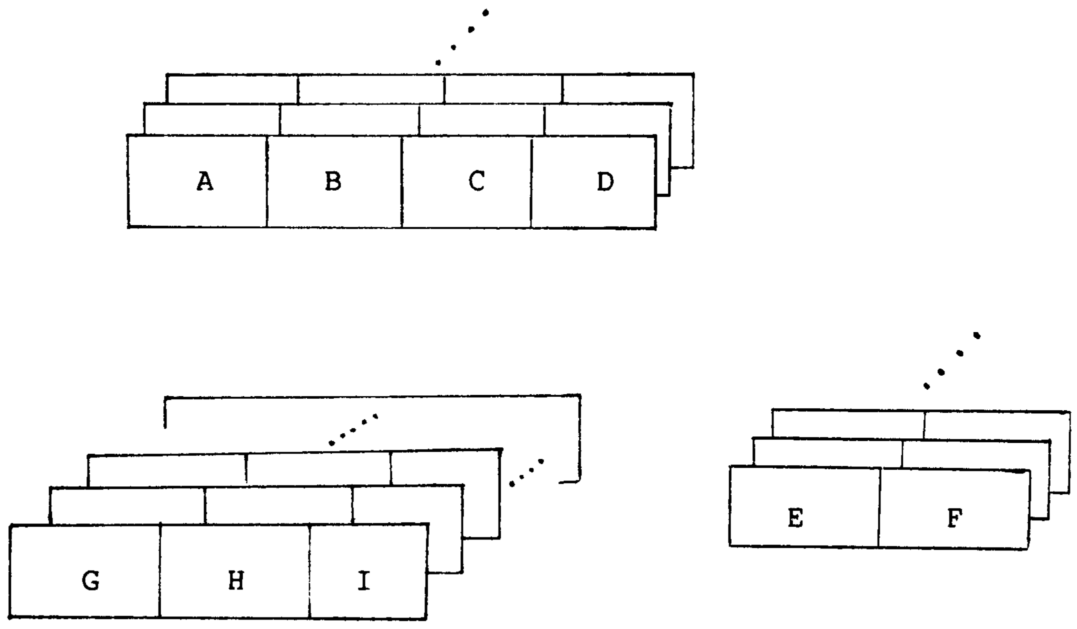
The hierarchical model of organising data is based on the notion of a tree: A hierarchical database model is a tree with records as its nodes.



In the network models, the organisation of data is not restricted to trees : A network model is a directed graph which has records as its nodes.



In the relational model the records of the same type are grouped together and pointers are removed.



The database in all these models is, therefore, a vast collection of records, and database management is the art of handling these records. The unhappy fact is that, traditionally, these structures are made visible to the user. With the first two models, the user has to navigate a path through the jungle of records. In a company database, for example, a simple query such as "name of the department in which Smith works" in Codasyl [O1-78] notation is:

```
EMPLOYEE.NAME='SMITH'.
FIND EMPLOYEE RECORD BY CALC-KEY .
FIND OWNER OF CURRENT ITEM RECORD SET .
GET DEPT .
PRINT DEPT.DNAME .
```

In the relational model the paths are eliminated, but the user still has to think in terms of records. As an example, we formulate the query "all employees who earn more than their managers" in Codd's sublanguage ALPHA [Co-71]:

```
RANGE  R1  X
RANGE  R2  Y
GET W (R1.EMPLOYEE): ∃X ∃Y (R1.DEPT=Y.DEPT) &
                          (Y.MANAGER=X.EMPLOYEE) &
                          (R1.SALARY > X.SALARY)
```

where R1 is a relation on EMPLOYEE, SALARY and DEPT, and R2 is a relation on DEPT and MANAGER.

Note that the variables X and Y range over records. Even recent proposals, such as the entity-relationship model [Ch-76], Aggregate model [SS-77] and SDM [HM-79], merely build additional structures over the traditional building blocks (i.e. records).

C. Problems of the Record Based Databases

Many authors now argue that records are unnecessarily complex and still inadequate [Ke-78 , Se-79], and that the database user should view the data in a less technical way.

We note that with the present technology, records are most suited for internal representation and processing. However, we argue that the user should not be aware of the internal structure of the database. We stated before that the user's view and the engineer's view need not be necessarily the same. Once this distinction has been made, we can then look for a more appropriate model of the real world for the user. "The more we are motivated to produce a faithful model of real information, the more we will have difficulty with record based constructs."[Ke-79]

One of the main reasons why database query languages merely reflect the internal model lies in the tendency of some database researchers to work in isolation and ignorance of new developments in other fields of Computer Science. We argue that database theory should not be separated from the other fields of Computer Science. In particular, database languages should have the same properties as those of other nonprocedural languages, i.e. values should be specified in an abstract mathematical way, with no reference to any particular implementation method. We explain this by means of an analogy:

Suppose that Fred wants to ask Joe to cut a metal disc of a certain diameter. There are several ways in which Fred can specify what is needed.

In the tedious way, Fred may (somehow) draw a circle (or something approximating to a circle) and give it to Joe, so he can use it as a pattern. Here, Fred anticipates Joe's work in a token way.
This approach, resembles formulating queries in COBOL style query languages. In such languages, the user finds his answer by directing a pointer through the collection of records.

Alternatively, Fred may specify the circle by its mathematical notion. Correspondingly, the user of nonprocedural languages would give an abstract specification of the objective.

We note, however, that the relational calculus query languages, although nonprocedural, reveal some of the operational aspects to the user and involve him (at least partly) in technicalities. In our analogy, it is as if Fred had to say that he wants a piece of metal in the shape drawn by a pair of compasses! We would criticize that Fred is burdened with details of how the work shall be done. It may be the case that a pair of compasses is the most appropriate tool, but why should Fred be concerned with that?

Records in the relational calculus languages, resemble the pair of compasses. Records may be good operational tools, but the user should not be aware of them.
(The next section of this chapter focuses on other aspects of the relational approach to databases.)

In the subsequent chapters we will demonstrate that the algebraic approach [Zi-75 , Gu-77 , GTW-78] provides what we are aiming for: independence from any concrete representation and from any technique for implementation.

D. Assessment of the Relational Approach to Databases

The relational theory of databases is based on relational algebra. Informally speaking, an algebra of relations is basically a collection of relations together with some operations which can operate on these relations.

A relation on a sequence of domains is any subset of the cartesian product over that sequence. For example, if the domains are $D_0, D_1, \ldots.$ and D_n then any subset of $D_0 x D_1 x \ldots . x D_n$ is a relation over D_0, $D_1, \ldots.$ and D_n. (There is an alternative method of definition for relations and it will be studied later.) Members of relations are called 'tuples'.

As relations are sets, the set theoretical operators, such as union and intersection can be applied to them. Other operations are:

concatenation ; (wrongly called "cartesian product" in the literature)

Given two relations R and S, their concatenation $R \chi S$ is

$$\{ r^\frown s \mid r \in R \text{ and } s \in S \}$$

where $r^\frown s$ is the concatenation of r and s.

(Concatenation of two finite sequences r and s is a sequence t, such that:

i) $|t| = |r| + |s|$

ii) $t ! |r| = r$ and $\forall j \in |s| \; t_{|r|+j} = s_j$.

For any sequence d, $|d|$ is the length of d.)

projection ; If R is a relation over a sequence of domains D, then the projection of R over i, where i is a sequence of natural numbers less than |D|, is

$$\Pi_i(R) = \{ (r_{i_0}, r_{i_1}, \ldots) \mid r \in R \}$$

selection ; Given a relation R over a sequence of domains D, and a formula F built up from:

- relation symbols = , > , $\geq$,....which operate on elements of |D| or constants
- the logical operators & , v and $\neg$,

the selection of R over F, denoted by $\sigma_F(R)$, is the set of all elements of R which satisfy F.

Based on these operations several other operations such as join and quotient are defined. Relations are often represented as tables. Thus, projection and selection correspond to choosing columns and rows from a relation respectively.

All counterpart definitions in the available literature make use of a notion which, when relations are defined in this manner, cannot be defined; namely `arity` of relations. Arity of a relation is defined to be the length of its sequence of domains. For example, if R is a relation on D_0, D_1,.... and D_n, the arity of R is said to be n+1. Some argue that although when the relation is not empty the above statement is obvious, however, the arity of the empty relation is undefined. Note that we did not make use of the notion of arity in our definition of relational algebra. This demonstrates the redundancy of the use of arity.

The immediate issue arising from this type of specification is that it associates a number with each column of the relation, that is, the columns are identified by a sequence of numbers. Consequently, most relational algebra expressions involving projection (which are commonly accepted) are not mathematically clear. For instance, if R is a relation on domains D_0, D_1, D_2 and D_3 then the following expression is not meaningful

$$\sigma_{3=\text{"abc"}} \Pi_{0,3}(R)$$

because projection of R over columns 0 and 3 results in a binary relation which obviously does not have a column corresponding to 3 (since after the projection the columns are renumbered).

To eliminate the ordering on columns, among others,[SY-78 , ASU-79] made alternative definitions for relations. They regarded a relation on a set of attributes as a set of tuples where each tuple is a mapping from the set of attributes to the set of values (attributes are considered to be labels, i.e. names). Tuples are then sets and their elements are not ordered. (Incidently , a misuse of notation repeatedly appears in the literature adopting such a definition: the attribute (i.e. a name) is commonly used also to denote the set of its corresponding values.)

The relational database work has overlooked a number of other points. We will, however, merely look at two more fundamental weaknesses of this theory.

The first problem is in the design of relational databases. The dependencies which exist on the data cannot be expressed in relational algebra. There are various types of such dependencies; namely, functional, multivalued and join dependencies. The database designer should look for all types because these dependencies play major roles in the maintenance of consistency (that is, correctness) and in reducing the size of the database. The study of data dependencies has resulted in production of a great deal of literature which is perhaps more than the literature on other aspects of relational database theory. Yet, no suggestion is fully satisfactory; [BB-79 and among many others Fa-77 , Bi-78 and Fa-81].

Secondly, the traditional theory of relational algebra assumes that the data has a tabular form, and, therefore, makes no provision for computable relations. For example, suppose, in a company, the salary of each employee is determined solely by his age, say: 150 times his age. Although this relationship is simple and straightforward, it still has to be viewed as a table. We, therefore, argue that the relational approach is limited because it enforces a particular method of representation for relations.

In chapter IV we will discuss the weakness of relational database query languages for expressing certain types of queries.

E. New Trends in Computer Science

Since the early 1960's when the first high level programming languages were invented, the search for more powerful languages has steadily continued. The number of existing languages is so large that the introduction of a new language gives no immediate cause for interest. The reason, perhaps, is that the similarities between these languages are greater than the differences. [Tu-81] indicates that these differences are usually superficial, whereas the similarities are fundamental. At a certain level of abstraction all conventional programming languages are the same:

- they have the inherent defect of having the von Neumann computer as their common conceptual origin. [Ba-78]
- they are sequential and imperative (due to the nature of the machine), and assignment is their primary operation.
- they cannot easily relate to conventional mathematics because their variables are not static within their scopes
- above all, reasoning about correctness of programs written in these languages is difficult and often totally impractical.

In an attempt to remedy the crisis, in the past decade, mathematically conscious Computer Scientists have tried to formalise the art of design and analysis of programming languages.

It is now widely accepted that mathematics, particularly logic can be usefully employed for the study of computer languages. Ashcroft and Wadge

[AW-79], however, distinguish two points of view on the application of mathematics: one which sees mathematics as a tool to describe, to model or to analyze programming languages (the descriptive role), and the other, which sees mathematics as "playing primarily an active role to discover the principles on which new languages and features should be based" (the prescriptive role).

Production of a vast amount of highly sophisticated mathematics for the description of languages such as PL/1 is an example of approach with the first point of view, while some other languages, such as PROLOG [Ko-79], Lucid [AW-77] and SASL [Tu-81] are based on mathematics in a prescriptive role.

In this thesis we take the second point of view and define a query language based on the notation used in conventional mathematics. This method has the advantage of making use of the user's mathematical knowledge rather than introducing new concepts which are inherently complicated and often counter-intuitive.

F. Motivation for Varqa

The perspectives in the development of Varqa can be summarised as:

1 . The query language should provide the tools by which the user can get the best results from the database without having to go through masses of printout.

2 . A meaningful, easy to understand and rigorous 'error analysis' should be mandatory with any computer language .

3 . The user should have the simplest view (which is obviously the natural one) of the data. The user should not be required to know anything about the internal representation and implementation techniques.

4 . The language should be independent from the data structure: the internal data model should not affect the language.

5 . To avoid dealing with the natural ambiguity of words , the use of mathematics which has the same meaning in all contexts is preferred. Correctness can be proved in formal systems.

6 . It is to the benefit of the user if the notation of the language is simple. Most users are familiar with conventional mathematical notation. This notation, in addition, has several hundred years of testing and development behind it.

7 . Finally, implementation of the whole specified language must be possible. The feasibility of implementing all of the features of the language is considered in this work.

Chapter II

Chapter II

AN OVERVIEW OF THE LANGUAGE

This chapter contains an informal introduction to the major concepts and constructs of Varqa.

A. Objects in the real world and objects in a database

Extensional objects can be loosely defined as follows: conceptual, static objects, such as integers, which are invariant with time, place or context etc. Intensional objects, on the other hand, are objects of the real world which are capable of changing and yet in essence remaining the same .

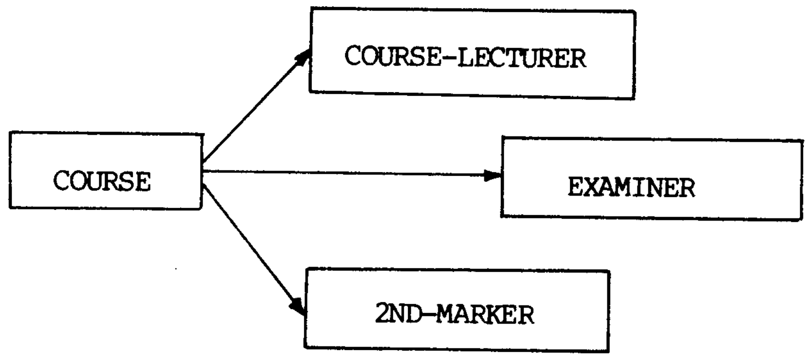
For example, TEMPERATURE is intensional, but TEMPERATURE AT A SPECIFIC TIME is extensional assuming that we know where and how this value was obtained. (Montague's example: "The temperature is 90 and rising".)

In the field of databases, particularly relational databases, objects are coded up intensionally. Let us consider a simple database consisting of one relation on information about courses:

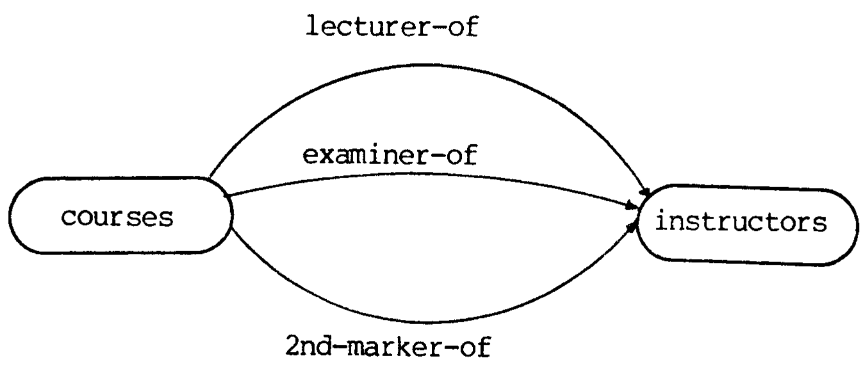
COURSES

COURSE	COURSE-LECTURER	EXAMINER	2ND-MARKER
"Maths"	"John"	"Jack"	"Bill"
"Phys"	"Jack"	"Mary"	"Pam"

Note that the column headings, called relation schemes, are intensional objects and the entries to the columns are extensional objects. There are 'functional dependencies' which cannot be expressed in the relational form. They are usually given in the following form:



Although LECTURER , EXAMINER and 2ND-MARKER differ intensionally, they range over a common domain, namely the set of all instructors. The extensional approach is to forget about different intensional characteristics of the instructors, and to deal with them just as members of a set (of instructors). The relationship between the extensions would then be expressed by functions mapping these sets together.



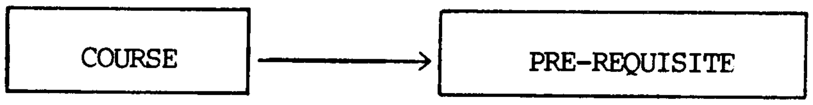
It is important to note that 'courses' is now a set, and that it is different from the intensional object COURSE in the relational approach.

Let us look at another example. In the relational approach, if we want to include the `prerequisites of the courses` in the database we have to introduce a new relation R on COURSE and PRE-REQUISITE. (We can extend the relation COURSES to include PRE-REQUISITE, but it would not satisfy the normal forms of Codd. See [Co-71].)

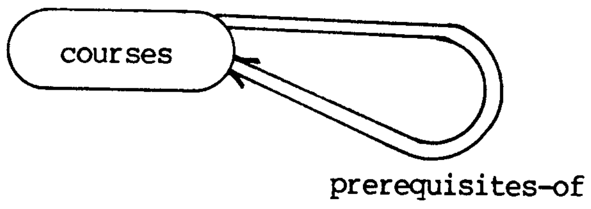
R

COURSE	PRE-REQUISITE
"Phys-II"	"Phys-I"
"Maths-II"	"Maths-I"
"Phys-III"	"Phys-II"

The hidden functional dependency contained in R is made explicit in the following figure



Extensionally, the entries to both columns of relation R belong to one set; that is 'courses'. An extensional representation is



(We have used double lines to indicate that for every course any number of prerequisites is possible.)

As before, 'courses' is a set, and 'prerequisites-of' is a function which maps the elements of 'courses' to sets of elements of 'courses'.

B. Databases as Algebras

During the past decade algebra has emerged as a promising tool for the specification of a number of concepts, in particular, abstract data types [Zi-75 , Gu-77 , GTW-78]. Algebra also plays a fundamental role in the design of our language. We, therefore, start this section by studying algebras and the algebraic approach.

An algebra is defined as follows [HMT-71]:
"By an algebra (or an algebraic structure) we understand a pair $\mathfrak{U}=\langle A,Q\rangle$ where A is a non-empty set and Q is a function which correlates with every element i of its domain a finitary operation Q_i, of positive rank, on and to elements of A." For a more detailed definition of algebras see [MB-79 , GH-78].

Many sorted algebras have been particularly important in both the practice and the theory of the specification of abstract data types. The primary aim in the abstract data type specification is to (precisely) describe a data type independently of any representation of its data objects and independently of any implementation of the operations. [TWW-78]

A data type is regarded as a many-sorted algebra and is defined to be [GTW-78]: an indexed family of sets (called the 'carriers') together with an indexed family of operations between these carriers. The naming system (i.e. indexing) is given by a 'signature'. The signature consists of a set S of names (called 'sorts') for the carriers, and a family

$\{ \Sigma_{w,s} \mid w \in S^* \;\&\; s \in S \}$ of operation names such that any $\varphi \in \Sigma_{w,s}$ with $w = s_0, s_1, \ldots s_{n-1}$ names an operation from $A^w = A_{s_0} \times \ldots \times A_{s_{n-1}}$ to A_s, where for any sort s, A_s is the carrier of sort s.

Other definitions for many-sorted algebras exist, e.g. [Ma-77] which are in harmony with the above. Further material on this topic can be found in, for example, [GTW-78 , Zi-75].

Following the ideas stated in the previous section, we regard a database as a collection of sets, called the "types" of the database, together with a collection of functions mapping the elements of these types together. The database is therefore a many-sorted algebra.

To the user, every function is merely a black box which when given a value, (possibly) returns a value. In other words, the user does not need to know about physical organization and representation of data, the ordering (that is, sorting) of sets, the positions of files or other technical aspects.

There is a name associated with every function and every type. These symbols are contained in a signature. In programming terms, the signature corresponds to type declaration for procedures . The signature contains the typing rules for the database mappings and also the types of variables. (The variables are typed by the signature and not by the user. There is an unlimited supply of variables of each type. We will later discuss the notion of variables in detail.) Hence, the signature is the specification for the type checker as well as for the syntax checker of the language.

The functions can return either a single data object or a set of data objects: we refer to them as simple functions and set-valued functions respectively. In general, these mappings are not everywhere defined. For example, let 'grade-of' be a function mapping 'students' and 'courses' to the set of integers (grade-of(X,Y) denotes the mark that the student X has achieved in the course Y). This function is obviously defined for certain pairs only, because not every student takes every course. We therefore introduce a new object Θ which stands for the value `not appropriate' or `inapplicable`. By adding Θ to our sets, our functions can be extended to be total functions.

Note that although we extend all our mappings, the extensions are hidden from the user: i.e. the user does not know anything about Θ other than as the result of a particular type of non-terminating query - they are queries which require applying a function to a value which is not in its domain. We will discuss the behaviour of Θ later in this chapter, and we will look at it again in chapter **V** when studying many-valued logic systems.

Example- Consider the function grade-of again. In extended form, it maps students$^+$ and courses$^+$ to integers$^+$. (For all sets D, D^+ is $D \cup \{\Theta\}$.) If the first argument given to this function is not of type 'students' or the second argument is not of type 'courses' , the type checker detects the error and evaluation does not take place. However, the value Θ is returned if the expression passes the type checker without the pair of arguments being in the domain of grade-of .

We treat constants, such as numerals, as nullary function symbols (functions which have no arguments). There are three types of nullary functions: numerals, strings of characters enclosed in quotes (e.g. "aB2K"), and reserved words which refer to distinguished objects in the database. For instance, in a university, we can regard the departments as distinguished objects and allocate them reserved words, e.g. The-Law-Dept for the Department of Law. Similarly a reserved word can be used to represent the `current` chancellor of the University. However, we would not allocate reserved words to ordinary objects such as the students or the courses, for the simple reason that they are subject to removal. (When a student leaves the university, the database normally does not keep any active information on him; if he has a reserved word, its deletion means an update to the language.) On the other hand, the office of the president of the student union is a permanent post of the university database and the current president of the student union can be represented by a reserved word, say The-SU-president.

Composition of functions is permitted, but instead of introducing the function composition operator, we allow functions to be applied to the results returned by other functions, i.e. f(g(x)) instead of f∘g(x). In this way the language is kept first order. Note that f∘g returns a function (i.e. a higher type object). For example, if f:B-->C and g:A-->B then f∘g is a function from A to C.

Full computational power is provided by including a wide range of operators in the language. The difference between operators and functions is that the operators form an unchangeable part of the language, whereas the functions are particular to the application database.

In addition to simple operators (such as arithmetic, boolean etc.) four variable binding operators are also included. They are the existential quantifier, the universal quantifier, the set constructor (e.g., $\{ f(x) \mid P(x) \}_x$) and the multi-set constructor (e.g. $[f(x) \mid P(x)]_x$).
In general, the operators are polymorphic although some, like + , operate only on specific types, e.g. integers (and of course Θ).

C. On Variables

What are variables?
There are several answers to this question: in traditional computer terminology a variable is an identifier attached to a register in the machine's memory; to logicians a variable is a certain kind of symbol; and mathematicians and physicists have again their own notion of variables. With the exception of elementary arithmetic, these disciplines employ variables very often. Here, we review first some given theories on variables, and then state our notion of variables.

Having investigated most concepts of this complexity, Russell, at the turn of the century, stated that there was not a satisfactory theory of variables in his Principles of Mathematics. Further, he admitted that this theory was certainly one of the most difficult to understand [Ru-03]. Russell considers anything which is not constant (i.e. absolutely definite) as variable. For instance, in the linear expression

$$Ax + By + C = 0$$

x and y are generally considered as variables, and A, B and C as constants; he argues that, unless we are dealing with an absolutely particular line, A, B and C are also variables.

Menger [Me-53] examines several theories on variables, such as that of Weierstrass. Given a class C of numbers, Weiertrass defined a numerical variable with the range C as a symbol standing for any element in C. For example, in

$$\frac{d}{dx} \ln x = \frac{1}{x}$$

the letter x is a variable whose range is all positive numbers. Menger sees this as inadequate and defines variable quantities in a complicated way as follows:

"Let A be any class. By a variable quantity with the domain A we mean a class of pairs such that: (1) in each pair the first element is an element of A, and the second is a number (called a value of the v.q.), (2) each element of A is the first member of exactly one pair belonging to the class. The class of all values is called the range of the v.q."

Curry and Feys [CF-58] see variables "as means of enunciating theorems about other things", the 'other things' being functions. The following are examples of theorems:

$$(x+1)^2 = x^2 + 2x + 1$$

$$x^2 \text{ is a function of } x$$

$$\frac{d}{dx} x^2 = 2x$$

The above statements, when interpreted, express some of the properties of the function SQUARE.

(Combinatory logic [CF-58] is concerned with the analysis of formal variables and their eventual elimination.)

Note that the above views are not contradictory : each one emphasizes a particular usage of variables. [AW-82] indicate more complicated cases where the variables can take on a whole family of ranges. Variable x in the following is an example:

$$\sum_{y=1,100} \prod_{x=1,y} x+y+1$$

For the purpose of this work, we take Weierstrass' concept of variables and generalize it to admit any set (not just numbers) as the range of a variable. However, we restrict our generalization by excluding variables which range over functions (in contrast to logicians who normally allow variables to stand for functions). In this way we can keep the language first order.

As customary in mathematics and logic, in Varqa we restrict our variables to certain sets : each of our variables has a type associated with it. For instance, in the statement "x is mortal" we require x to be a "living object" and not "anything".

An important and subtle point is that, although we regard variables as dynamic objects (things which change value with, say, time) we cannot talk about a specific permanent value for it. For example : if x stands for any integer, then saying 'x is 1' would be wrong. Because, in Russell's words, "it is not true that 1 is any number, though it is true that whatever holds of any number holds of 1". We, therefore, reject the traditional view of Computer Scientists' that variables denote storage cells. (Note that we can talk about the value of a variable in an environment. However, within an environment the value of a variable cannot change. In traditional Computer Science (imperative languages), the textual scope of the programs do not agree with the notion of environment.)

D. Variable Binding Operators

Let us compare two different ways in which variables can be employed. Often addition is defined in terms of the successor function as follows

$$x+y = \begin{cases} x & \text{if } y=0 \\ suc(x)+z & \text{if } y=suc(z) \end{cases}$$

On the other hand, sum of the possible values of an expression, say i^2+1, is expressed as :

$$\Sigma_{i=1,n} \quad i^2+1$$

In the above examples, although z and i are both variables, they have different characteristics: z acts as a placeholder, whereas i indeed ranges over the values 1 to n. The cause of this difference is the operator Σ which forces variable i to vary. There are many operators of this kind : we list a few examples which make use of them

$$\int_0^\infty \sin(x)\ dx$$

$$\forall x\ P(x)$$

$$\exists x>5\ Q(x,y)$$

$$\{f(x)\,|\,P(x)\}_x$$

$$\lim_{x \to \infty} \frac{x+1}{x^2+1}$$

These operators are called <u>variable binding operators</u> [KMM-80]. Each one of the symbols $\int$, $\forall$, $\exists$,.....above, denotes a variable binding operator.

In the simplest case, a variable binding operator takes a function as its argument and returns a value. Let f be a function mapping real numbers to real numbers. The value returned by

$$\int_a^b f(x)\, dx$$

is, in general, a real number. In abstract notation, the type of the denotation of $\int$ can simply be written as

$$[\mathbf{R} \dashrightarrow \mathbf{R}] \dashrightarrow \mathbf{R}$$

Similarly, in $\forall x\ P(x)$ where x is of type α, the type of the denotation of $\forall$ is

$$[\alpha \dashrightarrow \text{bool}] \dashrightarrow \text{bool}$$

where bool is {true , false}.

It must be stressed that, although functions are allowed (implicitly only) as the arguments of variable binding operators, we permit neither function-variables, nor function producing operators. It is our intention to keep the system in first order. (Some may argue that, by allowing operators which operate on functions, we have already departed from first order, but note that first order logic also makes use of quantifiers $\forall$ and $\exists$.)

E. The Extra Object θ

We start this section by justifying the inclusion of the special object θ. We also examine the alternative methods and state the problems involved in their use.

A naive way of representing the data is to designate particular types for the domains and ranges of functions. A case similar to this is observed in the relational model: There is a set associated with every attribute of the relational model (in relational database jargon: underlying domain). These sets do not need to be distinct. As a matter of fact, sometimes they overlap heavily : LECTURER , EXAMINER and 2ND-MARKER in the sample database given in the first section of this chapter are examples. This method of classification, however, in some cases becomes cumbersome and unnatural. Suppose the following table as part of a database:

X	sin(X)	cos(X)	tan(X)
0	0	1	0
30	0.5	.87	.58
45	.71	.71	1
60	.87	0.5	1.73
90	1	0	9999

Obviously, considering different types for sin(X) , cos(X) and tan(X) is not appropriate. In fact, we even do not have to consider a different type for X. Therefore, as suggested in [SS-77], similar objects are collected into (perhaps distinct) types.

When applying this abstraction to our suggestion (that is, collecting all objects of the same type into one big set), as discussed earlier, it is no longer true that the mappings are, in general, everywhere defined. There are two alternatives to put the situation right:

- we define subtypes within a type and we allow the functions to operate on subtypes only, or
- we introduce an "extra object" and thus extend the mappings into total mappings (Θ in our proposal is this extra object.) .

We examine these methods by means of examples.

Suppose in a company database the following information should be included

names of all employees

names of the spouses of the married employees

maiden names of all married women employees

The set of 'married women employees' is a subset of 'married employees', which itself is a subset of the 'employees'. We can represent the above information by total functions in the following form

```
name-of : employees ---> strings
spouse-name-of : married-employees ---> strings
maiden-name-of : married-women-employees ---> strings
```

It seems that the problem is solved and there is no need for the extra object. However, difficulty will be encountered when dealing with functions which take more than one argument. Earlier, we studied the function grade-of which operated on `students' and `courses'. To avoid dealing with partial functions, we have to consider a suitable subset of the set `courses' for each individual student (or vice-versa). (If there are two sets S and T where each one has three subtypes, then there are at least 9 subtypes of S**x**T.)

The problem is greater for functions of higher arities. In fact, the problem grows quickly in relation to the cardinalities of the sets and the arity. (Of course we can transform functions of greater arities to unary functions, but this does not solve the problem and, in addition, forces the use of higher order functions.)

We, therefore, find it more convenient to include the special object Θ.

Θ in our system is similar to the third value in three-valued logic systems, but is different from all of them in the way it behaves. (A detailed discussion on three-valued logic is included in chapter **V.**)
Θ can be interpreted as any of the terms 'misapplied' , 'inapplicable' or 'not appropriate'. It should, however, be made clear that it does not mean `inconsistent`, or `invalid`. Remember that faulty queries (i.e. those which encounter typing error) cannot be evaluated because they cannot pass the type-checker. Thus, we treat Θ as "does not apply to this individual", that is, <u>domain error</u>.

A function can return Θ either because

- it cannot have a value for the given input (This case is different from the case where a value should exist but is at present unknown.) or
- at least one of the arguments given to the function is Θ.

The operators have to be extended to handle Θ. An important rule governs the extensions of operations: the extended form in the absence of Θ must agree with the non-extended form, within the domain of the non-extended function. For example, the truth table of the operator 'and' is

	T	F
T	T	F
F	F	F

After the extension by Θ the table is

	T	F	Θ
T	T	F	Θ
F	F	F	F
Θ	Θ	F	Θ

Occurrence of Θ in the evaluation of a query does not imply that Θ is the final result because we make some exceptions. The result of applying any operation to Θ is Θ except for operators 'and' and 'or' which may stop Θ from propagating. For example, the operator 'or' will result in true so long as at least one of its operands is true. In addition, in the evaluation of queries involving variable binding operators, occurrences of Θ may be ignored. We will see that, for example, $\exists x\ P(x)$ is true if $P(x)$ is true for some value of x, it is false if $P(x)$ is false for all values of x, and it is Θ otherwise.

Finally, variables are not permitted to assume the value Θ. For instance, in $\forall x\ P(x)$ the value Θ is not allowed for x.

F. Queries

Queries are expressions which are built up out of function names, operation names and variables. (Data objects are denoted by nullary function symbols.) Only queries which satisfy the typing rules can be evaluated. Queries must be closed expressions; i.e. free occurrences of variables are not allowed in queries. Each variable can be bound only once in any expression; for example, expressions such as

$$\exists x\ ((x>5)\ \mathrm{v}\ (\forall x \in B\ x=2\ ))$$

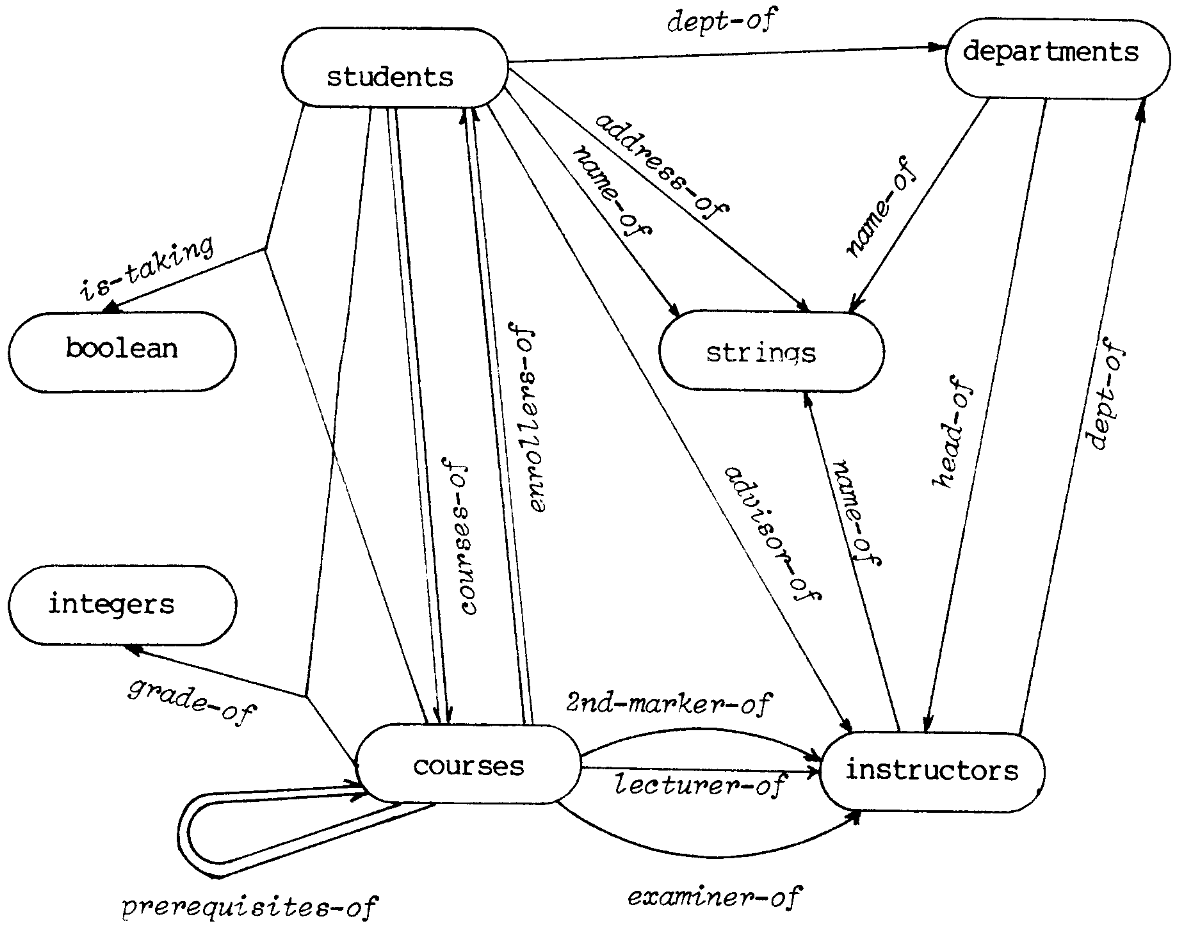
are not considered as valid queries. Note that these rules are all syntactic requirements.

The value of a query is defined by specifying the values which the algebra assigns to the parts and subparts of the query, in a fairly obvious way.

Queries cannot return higher order objects, i.e. functions. The values returned for queries can be data objects (as found in the database), sets of data objects, or sets of sets. There are several operators which operate on multi-sets of objects, such as 'Sum' and `Average`. Multi-sets of data objects are therefore considered. (A multi-set is analogous to a set but elements may appear more than once , e.g. [b,a,b,c,a,a].)

We now formulate a number of queries to demonstrate the power of our language.

Consider a university database, with several types such as students, courses, instructors, etc., and the functions lecturer-of, dept-of, courses-of, etc. In the figure below, the types appear in round boxes and the functions appear with arrows. Double arrows indicate the set-valued functions. Note that some of these function symbols may refer to the same operational values. For instance, courses-of, enrollers-of and is-taking are three functions which represent exactly the same information.



1. second markers of the courses whose lecturer is also the examiner

{ 2nd-marker-of(C) | lecturer-of(C) is examiner-of(C) } C

In the evaluation of this query, a set is constructed containing the second markers of all those courses which satisfy the condition -- lecturer of the course is the same as examiner of the course -- while the variable C iterates over the elements of courses. (The appearance of C on the very right indicates the variable which is being bound by the set construction operator. Although that is obvious in this query , we abide by the mathematical rules to preserve uniformity.)

2. name and address of the students who have taken more than six courses

{ (name-of(STUDENT),address-of(STUDENT)) | No-of(courses-of (STUDENT)) GT 6 }STUDENT

The result of this query is a set which contains pairs of names and addresses of some students (That is, a binary relation on names and addresses). The operator <u>No-of</u> counts the number of elements in the set returned by courses-of(STUDENT), and <u>GT</u> is an arithmetic operator which compares the result of the counting with the number 6.

3. students who are taking a course with their advisors

{ STUDENT | exists COURSE : COURSE isin courses-of(STUDENT) and advisor-of(STUDENT) is lecturer-of(COURSE) }STUDENT

There are two variables in this query: STUDENT which is bound by the set construction operator, and COURSE which is bound by the existential quantifier <u>exists</u>. The logical connective <u>and</u> gives the conjunction of the two boolean values returned by <u>isin</u> and <u>is</u>. <u>isin</u> is the usual set membership operator.

4. all enrollers of the courses taught by the lecturers of the Department of Law.

```
Union { enrollers-of(COURSE) | dept-of (lecturer-of (COURSE)) is
                                               The-Law-Dept }COURSE
```

The operator <u>Union</u>, when given a set of sets, computes the union of all included sets. (Another operator,namely <u>union</u> , exists, and it is the usual set theoretical operator U). enrollers-of is a set-valued function and hence, the set construction operator in this query builds up a set of sets. Nested application (or, composition) of functions is carried out here: dept-of is applied to the result returned by lecturer-of. <u>The-Law-Dept</u> is a reserved word standing for the Department of Law.

5. Is Jim the name of the head of the Mathematics Department?

```
"Jim" is name-of(head-of(The-Maths-Dept))
```

The answer to this query is returned by the operator <u>is</u> , and it is `yes' or `no` (true or false respectively).

Finally, to give a comparison with other query languages, we formulate the well known query : "all employees who earn more than the managers of their departments" of a company database. (In the previous chapter this query was formulated in ALPHA.)

$\{ E \mid \text{salary-of}(\text{manager-of}(\text{dept-of}(E))) \text{ LT } \text{salary-of}(E) \}$ E

G. Comparison With Other Work

With COBOL style query languages (mainly used with hierarchical and network data models) the user has to navigate a path through a jungle of records. The relational approach resolves this by introducing relational calculus as a base for non-procedural query languages. Relational calculus, however, still requires the user to think in terms of records. For example, its variables range over records. (We do not suggest that records are inherently evil, because in some cases they are extremely helpful. However, having everything based on records is unrealistic and far from the user's intuition.)

Varqa preserves the advantage of non-procedurality, furthermore, it provides the user with a simpler view of data. The way that the variables are used in Varqa is exactly that of conventional mathematics to which most users are accustomed.

Varqa is independent of any particular data model and can be used with an implementation based on any of the existing models, or even a combination of them. This advantage is more noticeable if we realize that relational databases make no provision for computable relations because they are based on a specific method of implementation.

This work is not the first endeavour for the design of a functional query language. Schwartz [Sc-71] was the first to mention the abstract algebraic view of databases, but he never pursued this idea any further. His general purpose programming language SETL, however, has several features similar to Varqa.

In [BF-79 , BNF-81] a functional query language, FQL, based on Backus' FP is suggested. Although FQL provides a powerful formalism for expressing queries, its notation is very complicated and somewhat unnatural. The elimination of variables in FQL makes the language very different from Varqa. Four functionals (higher order functions) are provided for combining the functions which map the data types together.

In the early version of FQL, the user had to declare the types (input and output types) of each expression and type checking was then done statically. The new version [BNF-81] relieves the user from the need for specification of types, since type checking takes place at run time.

In contrast to FQL, Daplex [Sh-81] has been developed with emphasis on notation. Daplex, however, lacks a formal definition. Many of its expressions are mathematically meaningless. Furthermore, several obvious things are missing. For example, Daplex fails to deal properly with functions of arities greater than one: the grades obtained by the students in their courses are given by declaring the following function

```
DECLARE Grade(Student,Course) ---> INTEGER
```

This function, however, is not defined for all pairs of students and courses. To avoid the problem, another choice of declaration is given:

```
DECLARE Grade(Student,Course(Student)) ---> INTEGER
```

We find this statement totally ambiguous; 'Course' which was used as a name for data type in the first statement, appears as a function name in the second (surprisingly, in some queries 'Course' is used as a variable ranging over the elements of the type Course, while, when necessary, being used as a function name!). Furthermore, in the second statement where Course stands for a function which when given a student returns a set of courses, the meaning of the function Grade is not clear: does it return only one value for all of the given set of courses, or does it operate on each individual course and return a set of values. It seems that although the notation suggests the former, the latter is the intended meaning.

Finally, Course, which (when standing for a function) normally operates on students, is later defined to be the inverse of the function

```
Title : Course ---> STRING
```

which is

```
Course : STRING ---> Course.
```

(The statements are exactly copied from [Sh-81].)

Most of other recent work on databases is concerned mainly with the relational model.

Following the algebraic approach to abstract data types, [CPP-78] formally describe a database model by its signature (a set of typed operations) and its presentation (a set of algebraic equations). [VM-81] consider equations as special cases of clauses and present an algorithm to translate these equations into logic programs automatically. This work, however, is more related to the study of updates rather than the construction of a query language.

The aggregate model suggested in [SS-77] has a good degree of abstraction. There are two kinds of abstraction considered:

- aggregation by which a relationship between objects is regarded as a higher level object
- generalization which allows a set of similar objects to be regarded as a generic object.

All objects, namely: individual, aggregate and generic, are then treated uniformly.

A semantic data model is presented in [HM-79] with a degree of similarity to the entity-relationship model of [Ch-76]. A positive point in this model is its provision for multiple ways of viewing the same information.

There is a trend in current database work towards a more constructive use of first order predicate logic for both data description [Ko-79] and construction of query languages [Ko-81 , Pi-78]. See [GM-78 , GMN-81] for comprehensive discussions.

Chapter III

Chapter III

FORMAL SPECIFICATION OF VARQA

In this chapter we define our proposed language in a formal manner. (In fact, we define not only one query language but a whole family of languages which essentially have the same structure. In this thesis, however, we consider only one of them, that is Varqa.)

Like other formal languages, the basic components of our language are "symbols"; with symbols we construct symbolic expressions. A symbol is a sequence of characters. The characters are contained in an alphabet. The set of symbols is called the 'vocabulary' of the language and is divided into four divisions: "type symbols" , "variable symbols" , "function symbols" and "operation symbols". We assume that the form of each symbol determines to which division it belongs.

For Varqa, we consider an alphabet consisting of:

- Roman letters; upper case and lower case
- numerals 0 - 9
- special characters; those available on an ordinary terminal, such as + | (< { : } ...

On this alphabet, we define four groups of symbols:

- type symbols

  These are strings of lower case Roman letters possibly followed by a numeral. A type symbol, however, cannot begin with the string "is" or end with the string "of".

- function symbols

  These are either:

  - non-nullary function symbols

    These are strings of lowercase Roman letters and possibly "-", which either

    - start with the string "is" or
    - end with the string "of" .

  - nullary function symbols

    These are either:

    - strings of Roman letters beginning with the string "The" and possibly containing "-", (we call these 'reserved words') or
    - any string enclosed in quotes (eg, "aB2K") or
    - strings of Roman letters and numerals possibly containing "_" which are neither type symbols,

function symbols nor reserved words or

- a string composed only out of the numerals 0-9.

- operation symbols

  These are either:

  - infix operators; They are:

    + - * / isin LT LE GT GE is is-not

    is-subset-of is-true-subset-of union intersection

    without and or implies.

  - prefix operators; They are:

    Union Intersection Sum Prod No-of Min Max

    Average not.

  - variable binding operator symbols; They are:

    forall exists {} [] .

- variable symbols

  These are non-empty strings of uppercase Roman letters possibly followed by any number of numerals.

When speaking meta-linguistically , we will use Greek letters to reason about the language.

A. Symbols and Symbolic Expressions

Definition: A simple type expression (ste) is a string of characters from our alphabet and the class of ste's is inductively defined as follows:

- type symbols are ste's
- if α_0, α_1,...and α_n, for some n, are ste's then $\alpha_0 U \alpha_1$, $(\alpha_0 x \alpha_1 x....x \alpha_n)$, $\mathbf{P}(\alpha_0)$ and $\mathbf{M}(\alpha_0)$ are ste's.

□

Remark:

Use of parentheses in $(\alpha_0 x \alpha_1)$ is necessary, because, in general, ((AxB)xC) is different from (Ax(BxC)).

Definition: A function type expression (fte) of arity n is a string of characters from our alphabet with the following form

$$\alpha_0,\ \alpha_1, \ldots\ldots, \alpha_{n-1} \text{ ----> } \beta$$

where $\forall i \in n$ α_i is a ste and β is a ste.

□

Definition: A signature is a function which assigns a function type expression to each non-nullary function symbol, and a type symbol to each variable symbol.

□

Remark:

Every signature must include the types 'bool' and 'int'.

Definition: Let Σ be a signature.

Let φ be a function symbol in the domain of Σ.

The arity of φ in Σ is the arity of $\Sigma(\varphi)$. (That is the unique n such that $\Sigma(\varphi)$ is a fte of the form $\alpha_0, \alpha_1, \ldots\ldots, \alpha_{n-1} \dashrightarrow \beta$ where $\alpha_0, \alpha_1, \ldots.$ α_{n-1} and β are ste's.)

□

Definition: A (many sorted) algebra is a function which assigns a set to each type symbol and a function to each function symbol.

□

Definition: Let **A** be an algebra.

Let α be a simple type expression.

The set of all objects of type α, denoted by $|\mathbf{A}|_\alpha$ is defined as follows:

- if α is a type symbol, then $|\mathbf{A}|_\alpha$ is the set of all objects of type α in the algebra **A**, that is, $\mathbf{A}(\alpha)$.
- if α is $\alpha_0 U \alpha_1$ then $|\mathbf{A}|_\alpha = |\mathbf{A}|_{\alpha_0} U |\mathbf{A}|_{\alpha_1}$.
- if α is $(\alpha_0 x \alpha_1 x \ldots x \alpha_n)$ then $|\mathbf{A}|_\alpha = |\mathbf{A}|_{\alpha_0} x \ldots\ldots x |\mathbf{A}|_{\alpha_n}$.
- if α is $\mathbf{P}(\beta)$ then $|\mathbf{A}|_\alpha = \mathbf{P}(|\mathbf{A}|_\beta)$.

- if α is $\mathbf{M}(\beta)$ then $|\mathbf{A}|_\alpha=\mathbf{M}(|\mathbf{A}|_\beta)$,

  where $\mathbf{M}(X)$ denotes multi-sets over the set X.

□

Definition: Let Σ be a signature.

Let $\mathbf{A}$ be an algebra.

$\mathbf{A}$ is a Σ-algebra iff for each function symbol φ in the domain of $\mathbf{A}$, if $\Sigma(\varphi)$ is α_0, α_1,.....,α_{n-1}----> β then $\mathbf{A}(\varphi)$ returns an element of $|\mathbf{A}|_\beta$ when given an element of $|\mathbf{A}|_{\alpha_0}$, an element of $|\mathbf{A}|_{\alpha_1}$, ... and an element of $|\mathbf{A}|_{\alpha_{n-1}}$.

□

Example:

Consider a signature Σ and a function symbol φ such that $\Sigma(\varphi)$ is α, β ----> γ.

If $\mathbf{A}$ is a Σ-algebra which assigns $\mathbf{Z}$ (the set of integers) to α and β, $\mathbf{R}$ (the set of reals) to γ, then given two integer numbers as the arguments, the function $\mathbf{A}(\varphi)$ will return a real number.

Definition: An <u>operation type expression</u> (ote) of arity n is a string of characters on our alphabet with the form

$$\alpha_0,\alpha_1,......,\alpha_{n-1} \text{ ---->}\beta$$

where $\forall i \in n$ α_i is a ste or a fte, and β is a ste.

□

Remark:

We assume that every operation symbol and every variable binding operator has an ote associated with it. The arity of the operation symbol is the arity of the ote associated with it.

Definition: Let Σ be a signature.

The set of all <u>well-typed expressions</u> on Σ, denoted by E_Σ is

$\bigcup E_\alpha$ for every type α

where E_α is the set of all well-typed expressions of type α and is inductively defined as follows:

- variable symbols of type α are in E_α.
- if φ is a function symbol such that $\Sigma(\varphi)$ is $\alpha_0, \alpha_1, \ldots, \alpha_{n-1} \dashrightarrow \alpha$ and $e_0 \in E_{\alpha_0}$, $e_1 \in E_{\alpha_1}, \ldots$ and $e_{n-1} \in E_{\alpha_{n-1}}$ then

  $\varphi(e_0, e_1, \ldots, e_{n-1}) \in E_\alpha$.

- if e_0 and e_1 are in E_α then

  e_0 is e_1 $\in E_{bool}$

  e_0 is-not e_1 $\in E_{bool}$

- if e_0 and e_1 are in E_{int} then

  e_0 GT e_1 $\in E_{bool}$

  e_0 LT e_1 $\in E_{bool}$

  e_0 GE e_1 $\in E_{bool}$

  e_0 LE e_1 $\in E_{bool}$

  $e_0 + e_1$ $\in E_{int}$

$e_0 - e_1 \quad \in E_{int}$

$e_0 * e_1 \quad \in E_{int}$

$e_0 / e_1 \quad \in E_{int}$

- if $e_0 \in E_{\alpha}$, $e_1 \in E_{\mathbf{P}(\alpha)}$ and $e_2 \in E_{\mathbf{P}(\alpha)}$ then

  e_0 isin e_1 $\in E_{bool}$

  e_1 is-subset-of e_2 $\in E_{bool}$

  e_1 is-true-subset-of e_2 $\in E_{bool}$

  e_1 union e_2 $\in E_{\mathbf{P}(\alpha)}$

  e_1 intersection e_2 $\in E_{\mathbf{P}(\alpha)}$

  e_1 without e_2 $\in E_{\mathbf{P}(\alpha)}$

- if $e_0 \in E_{\mathbf{P}(\mathbf{P}(\alpha))}$ then

  Union e_0 $\in E_{\mathbf{P}(\alpha)}$

  Intersection e_0 $\in E_{\mathbf{P}(\alpha)}$

- if e_0 and e_1 are in E_{bool}, and X is a variable symbol then

  e_0 and e_1 $\in E_{bool}$

  e_0 or e_1 $\in E_{bool}$

  not e_1 $\in E_{bool}$

  exists X:e_0 $\in E_{bool}$

  forall X:e_0 $\in E_{bool}$

- if $e_0 \in E_{\alpha}$, $e_1 \in E_{bool}$ and X is a variable symbol then

  $\{e_0\}X$ $\in E_{\mathbf{P}(\alpha)}$

  $\{e_0 | e_1\}X$ $\in E_{\mathbf{P}(\alpha)}$

$[e_0]X \quad \in E_{\mathbf{M}(\alpha)}$

$[e_0|e_1]X \quad \in E_{\mathbf{M}(\alpha)}$

- if $e \in E_{\mathbf{M}(\alpha)}$ then

  No-of $e \quad \in E_{int}$

- if $e \in E_{\mathbf{P}(int)}$ then

  Max $e \quad \in E_{int}$

  Min $e \quad \in E_{int}$

- if $e \in E_{\mathbf{M}(int)}$ then

  Sum $e \quad \in E_{int}$

  Prod $e \quad \in E_{int}$

  Average $e \quad \in E_{int}$

- if $e_0 \in E_{\alpha_0}$, $e_1 \in E_{\alpha_1}$ and α is of the form $\alpha_0 U \alpha_1$ then

  $e_0 \quad \in E_{\alpha}$

  $e_1 \quad \in E_{\alpha}$

- if $e_0 \in E_{\alpha_0}$, $e_1 \in E_{\alpha_1}$,..... and $e_n \in E_{\alpha_n}$ then

  $(e_0, e_1, \ldots, e_n) \quad \in E_{\mathbf{P}(\alpha_0 x \alpha_1 x \ldots x \alpha_n)}$

- if $e \in E_{\alpha}$ then

  $(e) \quad \in E_{\alpha}$

□

Definition: Bound occurrences of a variable X in an expression is defined as follows:

- if the expression is of the form

  exists X:e

  or

  forall X:e

  where e is an expression

  then any occurrence of X in e is a bound occurrence.
- if the expression is of either of the forms

  {e}X [e]X

  where e is an expression

  then any occurrence of X in e is a bound occurrence.
- if the expression is of either of the forms

  $\{e_0|e_1\}X$ $[e_0|e_1]X$

  where e_0 and e_1 are expressions

  then any occurrence of X in e_0 and any occurrence of X in e_1 is a bound occurrence.
- if the expression e is of the form

  $\rho(e_0,e_1,\dots,e_{n-1})$

  where ρ is either a function symbol or an operation symbol and $\forall i\in n$ e_i is an expression

  then bound occurrences of X in e are those corresponding to bound occurrences of X in e_i for some $i\in n$.
- if the expression e is of the form

  $(e_0,e_1,\dots,e_{n-1})$

  where $\forall i\in n$ e_i is an expression

  then bound occurrences of X in e are those corresponding to bound occurrences of X in e_i for some $i\in n$.

□

Remark:

In the above definition, although we should have considered each operation symbol individually, we have used ρ to stand for all of them (as well as the function symbols). This is merely to avoid repeating the same thing over and over.

Definition: All occurrences of a variable in an expression which are not bound are said to be free occurrences.

□

Definition: A <u>closed expression</u> is an expression in which there are no free occurrences of any variables.

□

Remark:

We put a restriction on expressions that in any expression any variable can be bound only once.

Definition: A <u>query</u> is a closed expression in which any variable is bound only once.

□

Definition: An <u>environment</u> is a function which assigns (temporary) values (of the appropriate type) to variables.

□

Definition: A <u>database</u> is an ordered pair $< \Sigma , \mathbf{A} >$, where Σ is a signature and $\mathbf{A}$ is a Σ-algebra.

□

B. Semantics of the Language

The meaning of the expressions of our language is specified by inductively defining meanings for parts and subparts of the expressions. We first give semantics to function symbols and operation symbols, and then define meaning for the expressions.

Semantics of the function symbols

The algebra associates a function with every function symbol existing in its signature. The associated functions are in their extended form. We define the natural extension for functions as follows:

Definition: Let f be a function from $D_0 x D_1 x \ldots\ldots x D_{n-1}$ to D_n.
The natural extension of f, denoted by f^+, is a function from $D_0 U\{\Theta\} x D_1 U\{\Theta\} x \ldots\ldots x D_{n-1} U\{\Theta\}$ to $D_n U\{\Theta\}$ such that

$$f^+(x_0,x_1,\ldots,x_{n-1}) = \begin{cases} f(x_0,x_1,\ldots,x_{n-1}) \text{ if } \\ \qquad (x_0,x_1,\ldots,x_{n-1}) \in Dom(f) \\ \Theta \quad \text{otherwise.} \end{cases}$$

□

Semantics of the operation symbols

In the following we will use

a and b	as data objects
A and B	as sets of objects
W	as a set of sets of objects
X	as a variable.

The semantics of each operation symbol is given individually.

ARITHMETIC OPERATORS

"+"

+ associates with the operator

$\check{+}$: int U{Θ} x int U{Θ} --> int U{Θ}

where

$$a \check{+} b = \begin{cases} a + b & \text{if neither a nor b is } \Theta \\ \Theta & \text{otherwise} \end{cases}$$

($+$ is the ordinary arithmetic operator plus.)

The operations corresponding to "*" and "-" are multiplication and subtraction respectively, and are similar to the operation corresponding to "+".

"/"

The operator associated with / is

$\check{/}$: int U{Θ} **x** int U{Θ} --> int U{Θ}

where

$$a \check{/} b = \begin{cases} a/b & \text{if neither a nor b is } \Theta\text{, and b is not 0} \\ \Theta & \text{otherwise} \end{cases}$$

SET-THEORETICAL OPERATORS

"isin"

isin corresponds to

$\check{\in}$: αU{Θ} **x** αU{Θ} --> bool U{Θ}

where

$$a \check{\in} A = \begin{cases} a \in A & \text{if neither a nor A is } \Theta \\ \Theta & \text{otherwise} \end{cases}$$

"is-subset-of"

The corresponding operator is

$\check{\subseteq}$: **P**(α)U{Θ} **x** **P**(α)U{Θ} --> bool U{Θ}

where

$$A \check{\subseteq} B = \begin{cases} A \subseteq B & \text{if neither A nor B is } \Theta \\ \Theta & \text{otherwise} \end{cases}$$

"is-true-subset-of", $\check{\subset}$, is similar to "is-subset-of".

"union"

This symbol is associated with

$\check{\cup} : \mathbf{P}(\alpha)\cup\{\Theta\} \times \mathbf{P}(\alpha)\cup\{\Theta\} \dashrightarrow \mathbf{P}(\alpha)\cup\{\Theta\}$

where

$$A \check{\cup} B = \begin{cases} A \cup B & \text{if neither } A \text{ nor } B \text{ is } \Theta \\ \Theta & \text{otherwise} \end{cases}$$

"intersection", $\check{\cap}$, and "without" (i.e. set difference) are similar to union.

"Union"

The operator corresponding to Union is

$\bigcup : \mathbf{P}(\mathbf{P}(\alpha))\cup\{\Theta\} \dashrightarrow \mathbf{P}(\alpha)\cup\{\Theta\}$

where

$$\bigcup W = \begin{cases} \{x \mid x \in y \text{ and } y \in W\} & \text{if } W \text{ is not } \Theta \\ \Theta & \text{otherwise} \end{cases}$$

"Intersection"

This symbol corresponds to

$\bigcap : \mathbf{P}(\mathbf{P}(\alpha))\cup\{\Theta\} \dashrightarrow \mathbf{P}(\alpha)\cup\{\Theta\}$

where

$$\bigcap W = \begin{cases} \{x \mid \forall y \in W\ x \in y\} & \text{if } W \text{ is not } \Theta \\ \Theta & \text{otherwise} \end{cases}$$

COMPARISON OPERATORS

"GT"

$\check{>}$: int U{Θ} x int U{Θ} --> bool U{Θ}

where

$$a \mathbin{\check{>}} b = \begin{cases} a>b & \text{if neither a nor b is } \Theta \\ \Theta & \text{otherwise} \end{cases}$$

The operators $\check{\geq}$, $\check{<}$ and $\check{\leq}$ correspond to GE, LT and LE respectively. Their extensions can be modelled on that of GT .

"is"

The corresponding operator is

$\check{=}$: α U{Θ} x α U{Θ} --> bool U{Θ}

where

$$a \mathbin{\check{=}} b = \begin{cases} a=b & \text{if neither a nor b is } \Theta \\ \Theta & \text{otherwise} \end{cases}$$

The operator corresponding to "is-not" is the negation of that of "is".

LOGICAL CONNECTIVES

"and"

This symbol is associated with

$\check{\&}$: bool U{Θ} x bool U{Θ} --> bool U{Θ}

where

$$a \check{\&} b = \begin{cases} \text{true} & \text{if a and b are both true} \\ \text{false} & \text{if at least one of a and b is false} \\ \theta & \text{otherwise} \end{cases}$$

"or"

The associated operator is

$\check{\vee}$: bool U{θ} **x** bool U{θ} --> bool U{θ}

where

$$a \check{\vee} b = \begin{cases} \text{true} & \text{if at least one of a and b is true} \\ \text{false} & \text{if a and b are both false} \\ \theta & \text{otherwise} \end{cases}$$

"not"

$\check{\neg}$: bool U{θ} --> bool U{θ}

where

$$\check{\neg} a = \begin{cases} \neg a & \text{if a is not } \theta \\ \theta & \text{otherwise} \end{cases}$$

The operator associated with "implies" is $\check{\rightarrow}$, where $a \check{\rightarrow} b$ is equivalent to $\check{\neg} a \check{\vee} b$.

MISCELLANEOUS OPERATORS

"Sum"

The corresponding operator is $\check{\Sigma}$ (which operates on multi-sets):

$\check{\Sigma}$: **M**(int)U{θ} --> int U{θ}

where

$$\check{\Sigma} D = \begin{cases} \Sigma_{i \in D} \; i & \text{if D is not } \Theta \\ \Theta & \text{otherwise} \end{cases}$$

"Prod" is the name for the operator computing the product of the elements of a multi-set. "Average" is associated with the operation for finding the average value of the elements of a multi-set. "No-of" stands for the "element counter". "Max" and "Min" correspond to the operations for finding highest value and lowest value within the elements of a set (of integers) respectively. All these operators behave in a similar way to the operator $\check{\Sigma}$.

Semantics of the expressions

Let Σ be a signature.

Let **A** be a Σ-algebra.

Let $\mathcal{E}$ be an **A**-environment.

Let E be a closed expression on Σ.

The value of E in algebra **A** and environment $\mathcal{E}$ is inductively defined as follows:

- if E is a variable symbol then the value of E is the value assigned to it by the environment $\mathcal{E}$.
- if E is an n-ary function symbol φ with n arguments $t_0, t_1, \ldots$

$,t_{n-1}$ (i.e. $\varphi(t_0,t_1,....t_{n-1})$.) then the value of E is the result of applying the function which **A** associates with φ to the values of $t_0,t_1,....$ and t_{n-1} in algebra **A** and the environment $\mathcal{E}$.

- if E is an n-ary operation symbol ρ with n arguments $e_0,e_1,.....,$ e_{n-1} (i.e., $\rho(e_0,e_1,...,e_{n-1})$.) then the value of E is the result of applying the operation which is associated with ρ to the values of $e_0,e_1,...$ and e_{n-1} in algebra **A** and environment $\mathcal{E}$.

- if E is of the form exist X:e where X is a variable symbol and e is an expression (of type boolean) then the value of E is :
  - true iff e is true in some environment $\mathcal{E}'$ differing from $\mathcal{E}$ at most in the value assigned to X
  - false iff e is false in $\mathcal{E}$ and in all environments $\mathcal{E}'$ differing from $\mathcal{E}$ at most in the value assigned to X
  - θ otherwise.

- if E is of the form forall X:e where X is a variable symbol and e is an expression (of type boolean) then the value of E is :
  - true iff e is true in $\mathcal{E}$ and in all environments $\mathcal{E}'$ differing from $\mathcal{E}$ at most in the value assigned to X
  - false iff e is false in some environment $\mathcal{E}'$ differing from $\mathcal{E}$ at most in the value assigned to X
  - θ otherwise.

- if E is of the form { e }X where X is a variable symbol and e is an expression then the value of E is
  - Θ if e is Θ in some environment $\mathcal{E}'$ differing from $\mathcal{E}$ at most in the value assigned to X
  - the set containing the values of e in all environments $\mathcal{E}'$ differing from $\mathcal{E}$ at most in the value assigned to X otherwise.

- if E is of the form { e_0 | e_1 }X where e_0 and e_1 are expressions and X is a variable symbol then the value of E is
  - Θ if e_0 is Θ in some environment $\mathcal{E}'$ differing from $\mathcal{E}$ at most in the value assigned to X in which the value of e_1 is true.
  - the set containing the values of e_0 in all environments $\mathcal{E}'$ differing from $\mathcal{E}$ at most in the value assigned to X in which the value of e_1 is true otherwise.

- if E is of the form [e]X where X is a variable symbol and e is an expression then the value of E is
  - Θ if e is Θ in some environment $\mathcal{E}'$ differing from $\mathcal{E}$ at most in the value assigned to X
  - the multi-set containing the values of e in all environments $\mathcal{E}'$ differing from $\mathcal{E}$ at most in the value assigned to X otherwise.

C. A simple database

We study in this section a database which models part of a warehouse (storage) system. The items stored in the warehouse are parts. The warehouse is divided into locations in which items can be stored. Each kind of part may occupy several locations, but each location may contain only one kind of item. The capacity of every location is limited. Some parts may be able to substitute others.

Along with the above information, we are interested to store in our database

- name, weight, substitute parts, locations and the quantity stored in each location   for every part
- the capacity of every location.

Solution

The types in our database are: parts, locations and strings. Recall that the types integer and boolean are present in all databases. We choose the type symbols \`parts\`, \`locations\`, \`strings\`, \`int\` and \`bool\` for our types. Further, we designate function symbols to stand for the functions giving the relationships between the data. Here is a list of the function symbols together with their function type expressions:

- if E is of the form $[\ e_0 \mid e_1\ ]X$ where e_0 and e_1 are expressions and X is a variable symbol then the value of E is
  - Θ if e_0 is Θ in some environment $\mathcal{E}'$ differing from $\mathcal{E}$ at most in the value assigned to X in which the value of e_1 is true.
  - the multi-set containing the values of e_0 in all environments $\mathcal{E}'$ differing from $\mathcal{E}$ at most in the value assigned to X in which the value of e_1 is true otherwise.

place-of	parts ----> **P**(locations)
item-of	locations ----> parts
is-containing	locations , parts ----> bool
name-of	parts ----> strings
substitutes-of	parts ----> **P**(parts)
weight-of	parts ----> int
qty-of	parts , locations ----> int
capacity-of	locations ----> int.

Figure 1 illustrates these relationships.

We will also supply an unlimited number of variable symbols of each type. Although we have a good degree of freedom in the choice of variable symbols, we choose, for the benefit of the user, symbols which reveal the type. Here are suitable variable symbols of type parts:

P, PØ, P1, P2,......... and PART, PARTØ, PART1, PART2,...........

Similarly, for the type locations we use the following variable symbols:

L, LØ, L1, L2,......... , LOC, LOCØ, LOC1, LOC2,.......... and
LOCATION, LOCATIONØ, LOCATION1, LOCATION2,........

The algebra associates the set containing all data of a type with the appropriate type symbol. Assume in the following that the set corresponding to the type locations is:

{1_1 , 1_2 , 1_3 , 2_1 , 2_2 , 2_3 , 2_4 , 3_1} .

We denote this set by λ (i.e. the set λ contains all locations). See figure 2 for the other sets.

Each function symbol corresponds to a function. For instance, the function associated with the function symbol capacity-of is f_7 below:

$$f_7 : \lambda^+ \dashrightarrow \mathbb{Z}^+$$

A complete list of functions which correspond to our function symbols is given in figure 3.

We now formulate some queries and discuss their evaluation.

1 - What is stored in location 100_A ?

item-of (100_A)

This query does not pass the type checker of the system, because the argument given to 'item-of' is not of the type 'locations'. (type error)

2 - What is the name of the part stored in the location 2_1 ?

name-of (item-of (2_1))

In the evaluation of this expression, the result returned by the function corresponding to 'item-of' is given to the function associated with 'name-of'. The answer to the query is "nut".

3 - All parts which are heavier than 200 grams.

```
{ P | weight-of(P) GT 200 }P
```

The variable binding operator {}, while forcing variable P to range over the elements of the set associated with 'parts', constructs a set consisting of those parts which qualify the condition. The answer of this query is {it_4 , z_13}.

4 - Locations of those parts which can substitute for mk_3200.

```
{ places-of(PART) | PART isin substitutes-of(mk_3200) }PART
```

substitutes-of returns a set of parts. Each one of these parts may have several locations. The answer to this query is therefore a set of sets, that is : {{1_1,2_1},{3_1}} .

5 - How many locations are occupied by the substitutes of the part mk_3200?
(i.e. Count the locations appearing in the result of query 4.)

```
No-of( Union { places-of(P) | P isin substitutes-of(mk_3200)}P)
```

Given a set of sets, 'Union' computes the union of all included sets. The operator 'No-of' counts the number of elements of the computed result.

6 - What is the total weight in those locations which contain the item pt_12 ?

```
{ qty-of(pt_12,L) * weight-of(pt_12) | L is-containing pt_12 }L
```

Note that this simple query cannot be formulated in languages based on the relational calculus in a straightforward manner. These languages, in general, require a host language (e.g. COBOL) to provide the tools for such queries.

7 - All parts which have more than one location with a capacity more than 200.

```
{ PART | No-of( { LOCATION | (LOCATION is-containing PART) and
                 (capacity-of(LOCATION) GT 200) }LOCATION ) GT 1 }PART
```

This complex query can be viewed to be composed of two nested expressions.

```
        { PART | No-of ( L )  GT 1 }PART
```

where **L** is

```
{ LOCATION | (LOCATION is-containing PART) and
                              (capacity-of(LOCATION) GT 200) }LOCATION
```

Note that **L** cannot be evaluated as a query because it has the free variable PART.

8 - What is the level of stock on part z_13 ? (i.e. inventory on z_13)

```
  Sum [ qty(z_13,LOCATION) | LOCATION  is-containing  z_13 ]LOCATION
```

The answer 30 is returned.

9 - All parts which occupy all locations with capacities more than 50.

```
    { P | forall L:(capacity-of(L) GT 50) implies (L is-containing P))}P
```

The empty set is the answer to this query.

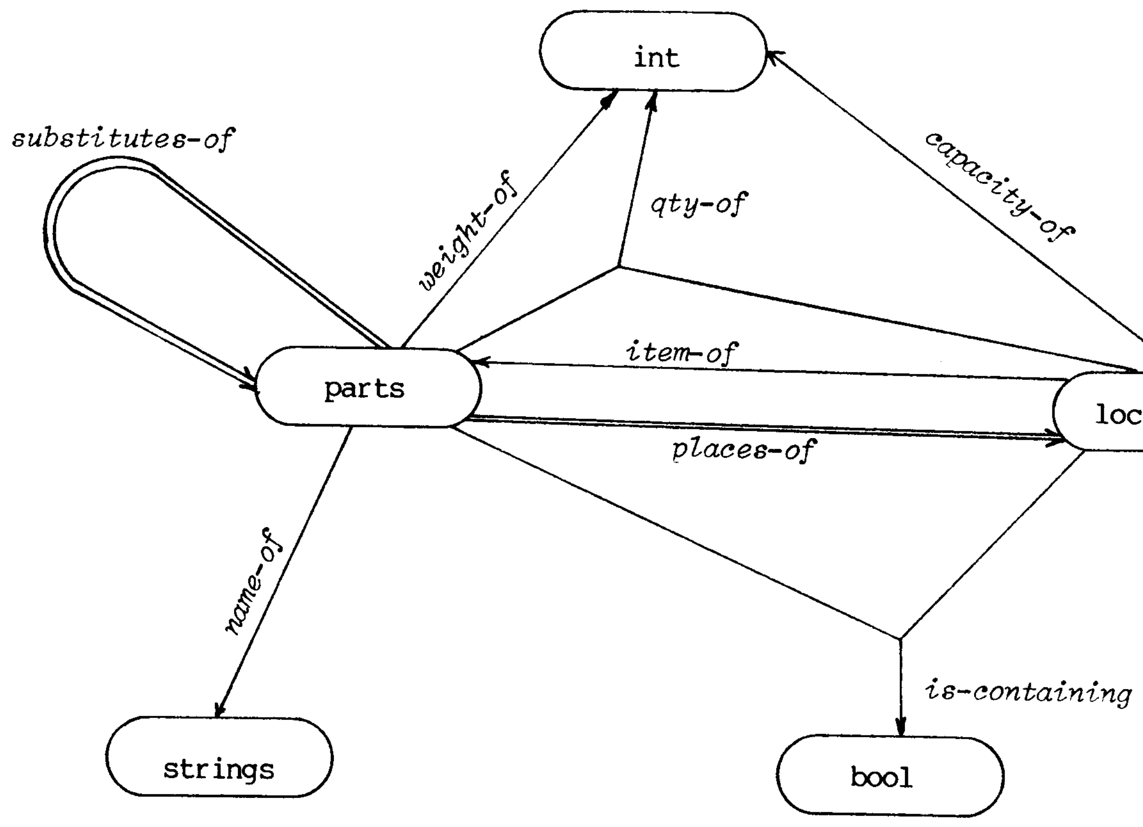


Figure 1 - Types and functions in a warehouse database

Figure 2

The type symbols and their associated sets in the warehouse database

Type symbol	Associated set
parts	p = { pt_12 , it_4 , mk_3200 , z_13 , pt_10 , z_100 }
locations	λ = { 1_1 , 1_2 , 1_3 , 2_1 , 2_2 , 2_3 , 2_4 3_1 , 3_2 }
strings	s = { "nut" , "bolt" , "washer" , "pipe" }
int	$\mathbb{Z}$ = { ....., -2 , -1 , 0 , 1 , 2 ,.....}
bool	B = { true , false }

(The types 'int' and 'bool' are present in any database.)

Figure 3

The warehouse database, functions and their values

places-of

$f_0 : p^+ \dashrightarrow \mathbf{P}(\lambda)^+$

{ <pt_12 , {1_1,2_1}> , <it_4 , {2_3}> , <mk_3200 , {1_2}> ,
<z_13 , {1_3,2_4}> , <pt_10 , {3_1}> , <z_100 , {2_2}> }

item-of

$f_1 : \lambda^+ \dashrightarrow p^+$

{ <1_1 , pt_12> , <1_2 , mk_3200> , <1_3 , z_13> , <2_1 , pt_12> ,
<2_2 , z_100> , <2_3 , it_4> , <2_4 , z_13> , <3_1 , pt_10> }

is-containing

$f_2 : \lambda^+ \times p^+ \dashrightarrow B^+$

This function has the same information as the two above.

name-of

$f_3 : p^+ \dashrightarrow s^+$

{ <pt_12 , "nut"> , <it_4 , "bolt"> , <mk_3200 , "nut"> ,
<z_13 , "washer"> , <pt_10 , "nut"> , <z_13 , "pipe"> }

substitutes-of

$f_4 : p^+ \dashrightarrow \mathbf{P}(p)^+$

{ <pt_12 , Ø > , <it_4 , Ø > , <mk_3200 , {pt_12 , pt_10}> ,
<z_13 , Ø > , <pt_10 , {pt_12}> , <z_10 , Ø > }

Figure 3 (continued)

weight-of

$$f_5 : p^+ \dashrightarrow \mathbf{z}^+$$

{ <pt_12 , 100> , <it_4 , 500> , <mk_3200 , 150> , <z_13 , 250> ,
<pt_10 , 100> , <z_100 , 20> }

qty-of

$$f_6 : p^+ \times \lambda^+ \dashrightarrow \mathbf{z}^+$$

{ <<pt_12 , 1_1> , 10> , <<pt_12 , 2_1> , 15> , <<it_4 , 2_3> , 50> ,
<<mk_3200 , 1_2> , 40> , <<z_13 , 1_3> , 20> , <<z_13 , 2_4> , 10>
<<pt_10 , 3_1> , 60> , <<z_100 , 2_2> , 30> }

capacity-of

$$f_7 : \lambda^+ \dashrightarrow \mathbf{z}^+$$

{ <1_1 , 10> , <1_2 , 60> , <1_3 , 20> , <2_1 , 60> , <2_2 , 40> ,
<2_3 , 50> , <2_4 , 60> , <3_1 , 70> , <3_2 , 5> }

Chapter IV

Chapter IV

INDUCTIVELY DEFINED SETS

A. Introduction

So far, we have demonstrated that our proposed language is a powerful tool for expressing a wide range of queries. However, there are yet other types of queries that cannot be expressed in Varqa using the facilities presented so far. Let us study the university database again. Consider the function 'prerequisites-of'. This function, when given a course, returns a (possibly empty) set of courses which are the immediate prerequisites of the given course. Sometimes the query is not as simple as that. One may wish, for example, to find out all the courses that a student must have passed before taking a particular course, say `Advanced Digital Systems`. This query asks for the prerequisites of Advanced Digital Systems, the prerequisites of each one of the computed prerequisites, and so on for any prerequisite computed (transitive closure). Similarly, within a company database, it is often necessary to determine all employees who directly or indirectly report to a certain manager. The "least timeconsuming combination of flights connecting two cities" is a usual query in airline reservation systems. There are many other examples of queries which cannot be formulated by the present notation of Varqa.

One way of solving the problem is to include several powerful operators in the language by which queries such as above can be expressed. For instance, an operator may be introduced which computes the transitive closure of any set-valued function. Such an operator, in the simplest form, takes a set-valued function and a value (i.e. TC(f,r) where f is a set-valued function and r is a value from the domain of f) and returns the set of all elements contained in the transitive closure. If f is of the form

$$A \dashrightarrow \mathbf{P}(A)$$

then the operator TC has the form

$$(A \dashrightarrow \mathbf{P}(A)) \times A \dashrightarrow \mathbf{P}(A) \; .$$

Using the TC operator, we can express the query "number of the courses which must be taken before taking Advanced Digital Systems" as follows:

No-of(TC(prerequisites-of , Advanced Digital Systems)) .

Although operators of this type are powerful, they do not give much versatility to the language. We include, therefore, a more versatile mechanism instead: we allow the user to define sets <u>inductively.</u>

By inductively defined sets we mean sets which are defined in terms of themselves, e.g. S=E(S) where E is some expression over the set S. This is a restricted form of recursion, because <u>we allow only sets to be recursively defined</u>. In ordinary functional programming languages, recursive definition of functions is usually permitted. A well known example is the recursive definition of the integer function for factorial:

```
fact(n):= if  n=0  then  0
                   else  nxfact(n-1)   fi .
```

However, as our primary aim is to construct sets, we restrict the recursive definitions to sets. The phrase "inductive definition" emphasizes this restriction.

For example, the transitive closure of the set-valued function f at b is expressed as

$$TC(f,b) = f(b) \cup (\bigcup \{ f(x) \mid x \in TC(f,b) \})$$

or, by using a `where` clause:

$$TC(f,b) = S$$
$$\text{where}$$
$$S = f(b) \cup (\bigcup \{ f(x) \mid x \in S \}) .$$

The value of such a definition is simply the least set satisfying the equation, that is, the smallest set containing all possible solutions.

The least solution is not always obtainable for definitions of the form S=E(S). For example, there are no least solutions for the definitions:

$S = T-S$ (Assumed T is not the empty set.)

$K = \{ x \mid x \notin K \}_{x \in K}$.

In this chapter we study the inductive definitions of sets. A detailed study of this area touches many other fields, and it is lengthy. We shall introduce only terms which are relevant in this thesis.

B. Related Work

Previous attempts to extend the non-procedural query languages to permit formulation of transitive closure are, again, based on the relational approach.

Query By Example (an informal system specifically designed for use with VDU's, for expressing queries by means of examples) took the first step by allowing the user to (indirectly) ask for the transitive closure of binary relations representing trees [Z1-75]. For instance, consider a binary relation on employees and managers; the managers themselves are employees and, in turn, have managers. In Query By Example, the user is enabled to specify a statement for the computation of : employees reporting to a certain manager at Nth level , or indeed all employees reporting to a particular manager. The formulation, however, is not direct. As there is no notion of least fixed point in Query By Example, the user has to specify the number of iterations required. This number must be large enough to succeed in the computation of the transitive closure.

A least fixed point operator has been imposed upon relational algebra in [AU-79]. This work defines the \`composition\` operator for two binary relations. (This operator has many similarities to \`equi-join\`. We argue against the definitions in this paper, because it contains the mistakes rejected in section D of chapter I.) The least fixed point operator is embedded in relational algebra in a seemingly ad-hoc manner. Transitive closure of a relation R_0 is then proven to be the least fixed point of the equation

$$R = R \circ R_0 \cup R_0$$

where $R \circ R_0$ is the composition of R and R_0.

It further proves that queries using the least fixed point operation cannot be expressed in a language which has the relational algebra operators only.

C. The WHERE Notation

`where` is common in ordinary mathematical notation. We list a few examples:

1) $x+a$

   where $x=b+2$

2) x^2+y^2

   where

   $x=a+b$

   $y=a-b$

3) $f(a)+f(b+1)$

   where $f(x)=x^2+1$

4) $\{ f(x) \mid x\in X \text{ or } x\in Y \}$

   where

   $X=\{0,1\}$

   $Y=\{ i^2 \mid i\in X \}$.

A phrase of the form

" where

definition(s) "

(that is 'where' followed by the definitions) is called a `where-clause`, and an expression of the form "expression where-clause" is called a `where-expression`. (Terms are from [La-66].)

Although we introduce the where-notation for formulating queries which require induction, we note that its use is not restricted to inductive definitions. For example, the query "all parts which have more than one location with a capacity of more than 200" of the warehouse database has been formulated as:

```
{P|No-of({L|(L is-containing P) and (capacity-of(L) GT 200)}L) GT 1}P
```

The alternative formulation is

```
{ P | No-of(S) GT 1 }P
    where
    S={L|(L is-containing P) and (capacity-of(L) GT 200)}L .
```

In addition to being elegant, this expression is easier to understand.

D. Inductive Definitions

For the non-recursive part of the language, we only needed variables which stand for the data objects of a certain type. Variables standing for sets of data objects are now required for inductive definitions of sets. Recall that each variable symbol has a specific type. `set variable symbols` are not exceptions: the type of a variable symbol standing for the set of objects of type α is $\mathbf{P}(\alpha)$.

Definition: Let Σ be a signature.

A `where-expression` on Σ has the form:

$$\begin{array}{l} e \;\; \text{where} \\ \qquad V_0=e_0 \\ \qquad V_1=e_1 \\ \qquad\quad . \\ \qquad\quad . \\ \qquad V_{n-1}=e_{n-1} \end{array}$$

when V_0, V_1,.... and V_{n-1} are variables, and e, e_0, e_1, ... and e_{n-1} are expressions on Σ, such that

i) $\forall\, i,j\in n \quad i\neq j \rightarrow V_i\neq V_j$

ii) $\forall i\in n$ the type of e_i is the same as the type of V_i.

□

Definition: Given a signature Σ and a Σ-algebra **A**, the value of a where-expression E of the form:

$$\begin{aligned} &e \ \text{ where} \\ &\quad V_0 = e_0 \\ &\quad V_1 = e_1 \\ &\quad \cdot \\ &\quad \cdot \\ &\quad V_{n-1} = e_{n-1} \end{aligned}$$

in an environment $\mathcal{E}$ is the value of e in an environment $\mathcal{E}'$ such that

i) $\mathcal{E}'$ is the same as $\mathcal{E}$ except for the values assigned to V_0, V_1,... and V_{n-1},

ii) the equations are all true in $\mathcal{E}'$,

i.e. $\forall i \in n \quad \mathcal{E}'(V_i) = \models_{\mathcal{E}'} e_i$

($\models_{\mathcal{E}} e$ is the value of e in environment $\mathcal{E}$.)

iii) $\mathcal{E}'$ is the least solution,

i.e. if $\mathcal{E}''$ satisfies i and ii then $\mathcal{E}'(V_i) \subseteq \mathcal{E}''(V_i)$ for all $i \in n$.

□

It is not difficult to determine the conditions under which the least solution can be obtained: we will use the least fixpoint theorem for complete lattices of [Ta-55] to reason about our expressions. Here is a summary of the relevant parts of this work :

A lattice is a system $\mathbf{L} = \langle A, \subseteq \rangle$ where A is a non-empty set and $\subseteq$ is a binary relation. The relation $\subseteq$ should establish a partial ordering on the set A. (A set P is partially ordered by the binary relation $\sqsubseteq$ if for any x, y and z in P the following properties hold:

i) $x \sqsubseteq x$ (reflexivity)

ii) $x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x=y$ (anti-symmetric)

iii) $x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$ (transitivity) .)

It is assumed that for any two elements $a,b \in A$ there is a least upper bound (that is $a \sqcup b$) and a greatest lower bound (that is $a \sqcap b$).

The lattice $\mathbf{L}=\langle A,\sqsubseteq\rangle$ is complete if every subset B of A has a least upper bound $\bigsqcup B$ and a greatest lower bound $\bigsqcap B$. In particular, a complete lattice has two unique elements $\bigsqcup A$ and $\bigsqcap A$.

Given a complete lattice $\langle A,\sqsubseteq\rangle$, a function f from a subset B of A to another subset C of A is called monotonic if for any two elements $x,y \in B$

$$x \sqsubseteq y \quad \text{implies} \quad f(x) \sqsubseteq f(y).$$

A fixpoint of a function f is, obviously, an element x of the domain of f such that $f(x)=x$.

Tarski proves that given a complete lattice $\langle A,\sqsubseteq\rangle$ and a monotonic function f from A to A, the set of fixpoints of f is not empty, further, there exists a least fixpoint for such a function (Theorem 1: lattice-theoretical fixpoint theorem of [Ta-55]).

We can easily demonstrate that Tarski's theorem is applicable to the expression $S=f(b)\cup(\bigsqcup\{f(x)\mid x\in S\})$ specifying the transitive closure of the set-valued function f :

The domain of the expression (i.e. powerset of a set) forms a complete lattice: the natural subset relation is the ordering, the operations union and intersection are applicable and we have the empty set as the least element.

Secondly, we demonstrate monotonicity for the function F, of the form $S=E(S)$ where $E(S)$ is $f(b)\cup(\bigcup\{f(x)|x\in S\})$.

Having the partial ordering $<\mathbf{P}(\alpha),\subseteq>$ we consider the chain

$$S_0 \subseteq S_1 \subseteq S_2 \subseteq \ldots\ldots \subseteq S_k$$

such that

$$\bigcup_{i=1,k} S_i \in \mathbf{P}(\alpha)$$

Given

$$S_0=\emptyset$$

we have

$$S_1=f(b)\cup(\bigcup\{f(x)|x\in S_0\}) = f(b)$$

$$S_2=f(b)\cup(\bigcup\{f(x)|x\in S_1\}) = f(b)\cup(\bigcup\{f(x)|x\in f(b)\})$$

$$=S_1\cup(\bigcup\{f(x)|x\in f(b)\})$$

$$S_3=f(b)\cup(\bigcup\{f(x)|x\in S_2\})$$

$$=f(b)\cup(\bigcup\{f(x)|x\in(f(b)\cup(\bigcup\{f(y)|y\in f(a)\})\})$$

$$=f(b)\cup(\bigcup\{f(x)|x\in f(b)\})\cup(\bigcup\{f(x)|x\in(\bigcup\{f(y)|y\in f(b)\})\})$$

$$=S_2\cup(\bigcup\{f(x)|x\in(\bigcup\{f(y)|y\in f(b)\})\})$$

.

.

$$S_{i+1}=S_i\cup(\bigcup\{f(x_0)|x_0\in(\bigcup\{f(x_1)|x_1\in(\bigcup\{\ldots\ldots\})\}\ldots)\})$$

□

As we are dealing with inductive definitions of sets only, proof of monotonicity is relatively simple. Apart from the set-difference operator, other set theoretical operators such as union and intersection are monotonic.

Given a variable S, based on the definition of formal monotonicity from [Pa-76 , Pa-82], the collection of expressions which are monotonic in S is inductively defined as follows:

- any expression not involving S is monotonic and anti-monotonic in S.
- S is monotonic in S.
- if the expression e is monotonic in S then any expression involving only e and any operation symbol other than "not" and "without" is monotonic in S.
- if the expression e is anti-monotonic in S then <u>not e</u> is monotonic in S.
- if the expression e_0 is monotonic in S and the expression e_1 is anti-monotonic in S then $\underline{e_0 \text{ without } e_1}$ is monotonic in S.

(φ is anti-monotonic iff $\forall A \; \forall B \;\; A \subseteq B \rightarrow \varphi(A) \supseteq \varphi(B)$.)

In other words, given any expression of the form S=E(S), E is monotonic if every occurrence of S in E is in an even number of distinct negated subparts of E.

Transitive closure of simple functions

The transitive closure of a simple function $f: \alpha \dashrightarrow \alpha$ may be specified as the least fixpoint of the equation:

$$S=\{f(b)\}\cup\{f(x)\mid x\in S\}$$

when b is a value in the domain of f.

This expression has the general form of $S=E(S)$, where the domain of E is $\mathbf{P}(\alpha)$. Since the only occurrence of S in E is not in a negated subpart of E, we conclude that E is monotonic. (This fact can be easily proved by a simple induction in the same way as we did before.) Therefore, there is a least set satisfying the above definition.

E. Evaluation in the Presence of Θ

Θ does not play any major role in the evaluation of an inductive definition of the transitive closure of set-valued functions. The reason lies in the way in which we have extended our set-valued functions. A set-valued function f maps a set A to the powerset of a set B. The extension of f is defined to be

$$f^{+}:A^{+}\text{-->}\mathbf{P}(B)^{+} \ .$$

f^{+}, when given a value, either returns the value Θ (iff the given value is Θ) or returns a (possibly empty) subset of B. (Note that, in general, $\mathbf{P}(B)^{+}$ is different from $\mathbf{P}(B^{+})$.)

The value of the expression

$$S=f(b)\cup(\bigcup\{f(x)\mid x\in S\})$$

is, therefore, Θ iff b is Θ.

Similarly, for the transitive closure of simple functions, the expression

$$S=\{f(b)\}\cup\{f(x)\mid x\in S\}$$

yields Θ when b is Θ. The computation will terminate when f does not have a corresponding value for the last computed value. For example, if f is

{<x5,x6> , <x7,x5> , <x8,x7> , <x9,x7> , <x10,x8> , <x11,x10>}

then the transitive closure of f from x11, TC(f,x11), is

{x10,x8,x7,x5,x6}.

Note that x6 is not in the domain of f.

Examples

We formulate two queries for the university database discussed in chapter II. Both of them require computation of the transitive closure of the set-valued function `pre-requisite-of`.

- lecturers of all those courses which must be taken before taking Adv-dig-sys.

```
{ (COURSE , lecturer-of(COURSE) | COURSE isin S }COURSE
     where
     S = pre-requisites-of(Adv-dig-sys) union
                         (Union{ pre-requisites-of(C) | C isin S }C )
```

- All lecturers who teach a course and one of its requirements.

```
{ I | exists C1 : (I is lecturer-of(C1) and
              exists C2 :(C2 isin S  and  I is lecturer-of(C2)) ) }I
     where
     S = pre-requisites-of(C1) union
                         (Union {pre-requisites-of(C3) | C3 isin S }C3)
```

Chapter V

Chapter V

INCOMPLETE INFORMATION

In this chapter we study the question "what are the consequences of allowing a database to contain incomplete data (i.e. the database has only partial information on some objects) ?".

Nonavailability of part of the data is a problem common to most database systems. In a university database, for instance , lack of information about the address of a student should not prevent the system from containing the (available) information on that student. However, admitting such a student into the database implies that 'no value' (or a value representing the missing value) be stored in place of the address of that student. We refer to these missing values as "unknown values", or "unknown" for short. (The term "null" has been used in the literature for any 'special value' , i.e. values other than the ordinary data objects.)

When unknowns are permitted in databases, a careful extension of the functions and the operations of the language is required to ensure that best results (i.e. answers which are consistent with reality and are closest to the true answer) are obtained.

Many-valued logic has been widely used for studying and reasoning about unknowns in databases. It will also be used in this work. Therefore, we first review some many-valued logic systems which are relevant to our work. Then, we study the null values that have been considered in the literature, and specify those values which are of interest to us.

Our principles for extensions will be defined in section C. The language will then be extended to handle incomplete information. We will see that the many-valued system devised based on our principles is different from the existing ones.

A. Many-Valued Logic, a survey

Although a relatively new branch of logic, many-valued logic has applications in a wide range of fields such as theory of partial recursive functions, modal logic and the study of semantical paradoxes. The basic distinction between many-valued logic and the classical two valued logic is, of course, that the former allows more than two truth values. (The term "pluri-valued" has been used to include both two-valued and many-valued systems. [Re-69])

Examples which call for more than the usual two values of truth and falsity are found when dealing with assertions about the future. Aristotle's discussions on "the occurrence of a battle tomorrow" are classic examples of this form. There are numerous other cases in which the values true (=T) and false (=F) are inadequate. In fact, in the previous chapters we dealt already with a 3-valued system, the third value being Θ. A number of other systems will be briefly reviewed here.

The 3-valued logic of Łukasiewicz : L3

Based on the arguments derived from Aristotle, Łukasiewicz simply extended the classical two-valued logic, C2, by an `intermediate` truth value I. This three valued system is called L3 [Re-69]. Using L3, he was able to reason about the matters relating to the future.

He extended the truth tables for the logical operators in the following manner (in fact, implication and negation were taken as primitives and the other operators were then defined in terms of them):

P	¬P
T	F
I	I
F	T

P->Q

P \ Q	T	I	F
T	T	I	F
I	T	T	I
F	T	T	T

P \ Q	T	I	F
T	T	I	F
I	I	I	F
F	F	F	F

P & Q

P \ Q	T	I	F
T	T	T	T
I	T	I	I
F	T	I	F

P v Q

Lukasiewicz tried to preserve most principles of the C2 system in his L3 system. The guiding principles of L3 are:

- T, I and F have decreasing values of `truthfulness`.
- The truth value of the conjunction of two assertions is the falsest, and the truth value of their disjunction is the truest of the two

components.

- Negation of a statement results in a truth value opposite to the truth value of the statement.
- The truth values of P->Q and ¬PvQ are not the same, because the value true is assigned to I->I . (This ensures that P->P remains a tautology. A tautology is a formula that is always true, regardless of the values assigned to its components. We will see that there are formulae which normally are tautologies but which are not tautologies in L3.)

By assigning values 0, 0.5 and 1 to F, I and T respectively, the truth values for negation, conjunction and disjunction can be summarized as: ([P] denotes the value of P.)

[¬P] = 1- [P]

[P&Q] = min([P],[Q])

[PvQ] = max([P],[Q]).

Bochvar's 3-valued logic : B3

Bochvar [Ch-39] introduced the 3-valued system "B3". The third value in B3 is different from that in L3: instead of using the intermediate value, Bochvar used the value "undecidable", U.
U in B3 can be interpreted as any of "paradoxical", "inconsistent" or "meaningless". Truth tables for the connectives in B3 are:

P	¬P
T	F
F	T
U	U

P -> Q

P\Q	T	F	U
T	T	F	U
F	T	T	U
U	U	U	U

P v Q

P\Q	T	F	U
T	T	T	U
F	T	F	U
U	U	U	U

P & Q

P\Q	T	F	U
T	T	F	U
F	F	F	U
U	U	U	U

When the above tables are used, the concept of tautology becomes impossible, since the occurrence of a propositional variable with the truth value U, in any expression, implies that the whole expression takes the value U.

A variant of B3 which has been suggested by Moh Shaw-kwei [Mo-54] interprets L3 in the Bochvarian approach. He proposed that the truth value F be given to UvF, instead of U as suggested by Bochvar. This measure was taken to preserve more of the tautologies of C2.

Kleene's 3-valued logic : K3

Kleene's motivation for introducing a further 3-valued system, K3, was to give the third value the meaning "undefined" or "undeterminable". We will use K to represent this value. K3 differs from L3 in the truth table for implication [Kl-62]:

P	¬P
T	F
F	T
K	K

P -> Q

P \ Q	T	F	K
T	T	F	K
F	T	T	T
K	T	K	K

P v Q

P \ Q	T	F	K
T	T	T	T
F	T	F	K
K	T	K	K

P & Q

P \ Q	T	F	K
T	T	F	K
F	F	F	F
K	K	F	K

In Kleene's system P->Q is equivalent to ¬PvQ . Consequently P->P is not a tautology, neither is P<-->P. Kleene referred to the above connectives as "strong" connectives. He then defined another set of truth tables, which were actually the same as those of B3, and designated them for "weak" connectives.

Generalisations of 3-valued logics

There are numerous many-valued systems developed by combining the 3-valued systems. The simplest generalisation was that of Łukasiewicz. He simply introduced more intermediate truth values. He retained his evaluation rules and, as before, allowed numerical evaluation for the logical connectives. For example, in his 4-valued system (L4) with I1 and I2 as the intermediate values, he assigned 1, 2/3, 1/3 and 0 to T, I1, I2 and F respectively. The values of expressions are obtained by the following arithmetic rules:

$$[\neg P] \equiv 1-[P]$$

$$[P \vee Q] \equiv \max([P],[Q])$$

$$[P \& Q] \equiv \min([P],[Q])$$

$$[P \rightarrow Q] \equiv \begin{cases} 1 & \text{if } [P] \leq [Q] \\ 1-[P]+[Q] & \text{if } [P] < [Q] \end{cases}$$

Łukasiewicz generalised his system to "infinite-valued" by taking negation and implication as primitives and by using the rules:

$$P \vee Q \equiv (P \rightarrow Q) \rightarrow Q$$

$$P \& Q \equiv \neg(\neg P \vee \neg Q).$$

However, the more interesting ones among the many-valued systems are those which yield L3 when reduced to the 3-valued case but are not generalisations of Łukasiewicz' system. For example, the 4-valued system with I and U (that is, the combination of B3 and L3) is the most popular one. Later, we will review the use of such a system in the theory of databases.

Other many-valued systems are more complicated but beyond the scope of this thesis. A detailed study can be found in [Ac-67 , Re-69].

B. Null Values in Database Systems

There are many cases in which a value in a database can be thought of as null. In previous chapters, we used Θ as the result of applying a function to an argument which is of appropriate type but is not in the domain of the function. In concrete cases, there may be various kinds of null values. For example, the function 'husband-of' returns the name of the husband of the married-women-employees. Whenever this function is applied to an argument, the following cases can be distinguished:

i) type-error : the argument is completely unknown to the function, (e.g. asking for husband-of(book)). This case corresponds to Bochvar's third value. We eliminated the evaluation of this situation by introducing the preliminary type checking.

ii) the type of the argument is correct but the function cannot have a value for it (for example, husband-of(Mary) when Mary is not married). Θ is the result value in our language (domain-error).

iii) the argument is in the domain of the function but the corresponding value is at present 'not available' (e.g. husband-of(Jane) where Jane is married but the name of her husband is not known to the system).

iv) the function returns a person (e.g. John for husband-of(Sally)).

There are several situations which would be candidates for case iii, for example: the value is not known at present (i.e. missing), the value is not permitted to be stored,or the value is undergoing change. [ANSI-75] lists ten situations where case iii applies.

Previous work on null values

Most of the work on null values has been carried out within the study framework of relational databases. Codd explained in [Co-75 , Co-79] a simple method for the treatment of missing values. Based on the 3-valued logic of Łukasiewicz, he extended the operators of the relational algebra to handle missing values.

Grant [Gr-77] found Codd's approach inadequate and demonstrated cases for which the extensions did not work.

In a more formal manner, Vassiliou [Va-79] uses a four-valued logical system to reason about nulls. His null values are "missing" and "nothing", which correspond to the third value in L3 and B3 respectively. (There are a number of mistakes in this work. For example, the rules for extending the functions do not cover all possible cases. See page 165 of [Va-79].)

There are also several other approaches which appear in the field of artificial intelligence. The concept of possible worlds in modal logic is the guiding principle for these systems. For example, Lipski and Imielinsky [Li-79 , IL-81], in a more rigorous approach, describe the semantics of a query by giving two different interpretations to it which they call _external_ and _internal_ interpretation. In the internal interpretation a query is evaluated by using only the knowledge available in the database. The external interpretation, on the other hand, refers directly to the real world modelled by the database rather than being confined to the incomplete information contained in the database. This work is mostly concerned with deduction of information and with artificial intelligence. We feel that this area is outside the scope of this thesis.

Another interesting study is that of Biskup presented in [Bi-81]. Two types of null values are considered. They correspond to "attribute is applicable but its value is at present unknown" and "attribute is applicable but its value is arbitrary". It is then assumed that the first type corresponds to an existentially quantified variable and the second type corresponds to a universally quantified variable. In Biskup's notation, if R is a relation on a given range D, then $(a,\exists,b,\forall)\in R$ means: there exists an $x\in D$ such that for all $y\in D$ the tuple $(a,x,b,y)\in R$. He extends the relational algebra operators and then, using predicate logic, proves correctness and completeness for the extensions.

Null values and Varga

In addition to θ, we introduce another object, δ, which represents the case that "the argument given to the function is in the domain of the function but the corresponding result is not available at the present time" (case iii of the example given at the beginning of this chapter). We feel that with these two extra objects most forms of null can be captured.

It would be helpful, of course, if the system could identify the reason for the non-availability of information, but this would require different symbols for the various manifestations of "unknown".

Further, although there should be a different symbol for the unknown information of each type (say δ_α for unknowns of type α) we use one symbol for all types, namely δ. In this way, the complexity of the underlying mathematics is considerably reduced.

It is important to note that each δ is a place holder for a separate value. We emphasize that the δ's are not, in general, equal (even those replacing values of the same type). For this reason we may have a set $\{\delta,\delta\}$ which may later result in a set consisting of either one or two objects (as missing information becomes available).

Semantically, despite being classified as null values, δ and θ are different. δ acts as a place holder for a fact which is presently unknown but which, perhaps, will be known at a later time. θ, on the other hand, stands for something which cannot be known at any time because it does not exist. When reasoning about predicates, δ can be interpreted as "true or false" (similar to the intermediate value of the L3 system), whereas θ

is independent from the other truth values. Hence, these two objects behave differently in the course of interpretation of queries.

Finally, we emphasize that Θ in our system disagrees with the third value of the Bochvarian three valued logic B3. This is due to the strict type checking prior to evaluation of queries.

Unknown sets

Since the functions in Varqa are permitted to return either a single value (i.e. a data object) or a set of objects , we should be able to reason about partially and totally unknown sets. Having δ, representation of partially unknown sets is possible. However, we cannot represent totally unknown sets. We therefore designate $\triangle$ to stand for a set of unknowns whose cardinality is also unknown. We will show later that this designation is considerably helpful in our formalism.

C. The Principle of Growing Certainty

We give the following principle for extending the query languages:

> Whenever information is added to an incomplete database subsequent answers to queries must not be less informative than previously.

Let us expand on this principle.

Suppose we have the sets D_0, D_1,.... and D_{n-1} in our algebra $\mathbf{A}$ (in ADJ's terms, the 'carriers' [GTW-78]) . Before allowing ignorance in the system, the universe of our algebra is

$$\mathbf{U}(\mathbf{A}) = W(\mathbf{A}) \cup \{\theta\}$$

where

$$\begin{aligned} W(\mathbf{A}) = \bigcup \{D_0, D_1, \ldots, D_{n-1}\} \cup \\ \{(T_0, T_1, \ldots, T_{m-1}) \mid \forall i \in m\ T_i \in W(\mathbf{A}) \text{ for some } m\} \cup \\ \mathbf{P}(W(\mathbf{A})) \cup \mathbf{M}(W(\mathbf{A})) . \end{aligned}$$

Permitting unknowns in the system extends the algebra. An extended algebra is like an ordinary algebra except the data operated on is in the extended universe. Our extended universe is slightly richer than a universe obtained by a straightforward extension. The straightforward method is to extend the universe to be:

$\mathbf{U}(\bar{\mathbf{A}}) = W(\bar{\mathbf{A}}) \cup \{\theta\}$

where

$W(\bar{\mathbf{A}}) = \bigcup \{\bar{D}_0, \bar{D}_1, \ldots, \bar{D}_{n-1}\} \cup$

$\{(T_0, T_1, \ldots, T_{m-1}) \mid \forall i \in m \; T_i \in W(\bar{\mathbf{A}}) \text{ for some } m\} \cup$

$\mathbf{P}(W(\bar{\mathbf{A}})) \cup \mathbf{M}(W(\bar{\mathbf{A}}))$

where $\forall j \in n \; \bar{D}_j = D_j \cup \{\delta\}$.

However, we also wish to include $\triangle$ in our extended universe. Therefore, we extend the universe as follows

$\mathbf{U}(\hat{\mathbf{A}}) = W(\hat{\mathbf{A}}) \cup \{\theta\}$

where

$W(\hat{\mathbf{A}}) = \bigcup \{\bar{D}_0, \bar{D}_1, \ldots, \bar{D}_{n-1}\} \cup$

$\{(T_0, T_1, \ldots, T_{m-1}) \mid \forall i \in m \; T_i \in W(\hat{\mathbf{A}}) \text{ for some } m\} \cup$

$\hat{\mathbf{P}}(W(\hat{\mathbf{A}})) \cup \mathbf{M}(W(\hat{\mathbf{A}}))$

where

$\forall j \in n \; \bar{D}_j = D_j \cup \{\delta\}$

$\hat{\mathbf{P}}(X) = \mathbf{P}(X) \cup \{Y \cup \triangle \mid Y \in \mathbf{P}(X)\}$ assuming $\delta \in X$.

$\mathbf{U}(\hat{\mathbf{A}})$ allows for more unknowns when compared with $\mathbf{U}(\bar{\mathbf{A}})$.

An algebra which contains unknown values cannot be considered as an exact representation of the real world ; it only "approximates" the real world. The algebras $\bar{\mathbf{A}}$ and $\hat{\mathbf{A}}$ above are approximations for **A.** We now say (only informally) what it means for an algebra to approximate another algebra. If A" is an algebra like A' except that the universe of A" has less information (i.e. it has more δ's and $\triangle$'s) then A" approximates A' or $A'' \sqsubseteq A'$.

Informally speaking, $a \sqsubseteq b$ when a can evolve into b as more information is made available. Given objects x and y, and sets A and B we have

$\delta \sqsubseteq \delta$

$\delta \sqsubseteq x$

$x \sqsubseteq x$

$\{\delta\} \sqsubseteq$ a set containing one object

$\underbrace{\{\delta,\delta,\ldots\ldots,\delta\}}_{n\ \delta\text{'s}} \sqsubseteq$ a set containing one, two,.....or n objects

$\triangle \sqsubseteq \triangle$

$\triangle \sqsubseteq A$

$A' \cup \triangle \sqsubseteq A$ where $A' \subseteq A$

$A \sqsubseteq A$

$A \sqsubseteq B$ where $\forall b \in B\ \exists a \in A\ \ a \sqsubseteq b$ and

cardinality of $B \leq$ cardinality of A

$(a_0,a_1,\ldots,a_{n-1}) \sqsubseteq (b_0,b_1,\ldots\ldots,b_{n-1})$

where $\forall i \in n\ a_i \sqsubseteq b_i$

$[a_0,a_1,\ldots,a_{n-1}] \sqsubseteq [b_0,b_1,\ldots,b_{n-1}]$

where $\forall i \in n\ \exists j \in n\ a_j \sqsubseteq b_i$ and $\forall i \in n\ \exists j \in n\ a_i \sqsubseteq b_j$

Thus, for two compatible algebras $\mathbf{A}'$ and $\mathbf{A}''$ we have

$\mathbf{A}'' \sqsubseteq \mathbf{A}'$ iff $\forall y \in \mathbf{U}(\mathbf{A}')\ \exists x \in \mathbf{U}(\mathbf{A}'')$ such that $x \sqsubseteq y$.

Our guiding principle for the extensions (<u>the principle of growing certainty</u>) can now be formalised as :

Given an environment, for any expression e and algebras $\mathbf{A}''$ and $\mathbf{A}'$

$\mathbf{A}'' \sqsubseteq \mathbf{A}'$ implies $e_{\mathbf{A}''} \sqsubseteq e_{\mathbf{A}'}$

where $e_{\mathbf{A}''}$ and $e_{\mathbf{A}'}$ are the values of e in algebras $\mathbf{A}''$ and $\mathbf{A}'$

D. Extending Varqa to Handle Null Values

Extension of simple functions

We have already extended our functions to total functions by including Θ in their domains and ranges. We now define a further extension for functions in order to enable them to operate on missing values. ($\hat{}$ will be used to denote the extended form of the functions: e.g. $\hat{f}$ is the extended form of the function f.)

Let $f:D_0U\{\Theta\}xD_1U\{\Theta\}x.....xD_{n-1}U\{\Theta\} \dashrightarrow D_nU\{\Theta\}$ be a function.
$\hat{f}:D_0U\{\Theta\}U\{\delta\}xD_1U\{\Theta\}U\{\delta\}x....xD_{n-1}U\{\Theta\}U\{\delta\} \dashrightarrow D_nU\{\Theta\}U\{\delta\}$ is a (further) extension for f iff $\hat{f}(x_0,x_1,....,x_{n-1})$ is

- $f(x_0,x_1,.....,x_{n-1})$ if $(x_0,x_1,....,x_{n-1})\in Dom(f)$

- δ if $\forall i\in n$ x_i is not Θ and x_j is δ for some $j\in n$

- Θ otherwise.

□

Example:

Consider a function "spouse-of" mapping "persons" to "persons". Let "persons" correspond to the set {John , Mary , Jane , Fred , Mark}, and "spouse-of" correspond to the function {<John , Mary> , <Jane , δ>}.

The following are examples of some possible cases:

i) spouse-of(John) is Mary

ii) spouse-of(Jane) is δ

iii) spouse-of(Fred) is Θ

iv) spouse-of(k) , where k is evaluated to be δ, is δ

v) spouse-of(k) , where k is evaluated to be Θ, is Θ.

Compare cases ii and iii. Jane is married but her spouse is not known, thus δ is returned. Fred, however, is not married; Θ is returned to indicate this fact.

(Again, evaluation will not take place if the argument is not of the appropriate type. That is,

spouse-of(k)

where $k \notin$ {John , Mary , Jane , Fred , Mark}

will not proceed to evaluation.)

Extension of set-valued functions

One way of extending a set-valued function of the form

$$f:D_0 \cup \{\Theta\} \times D_1 \cup \{\Theta\} \times \ldots \times D_{n-1} \cup \{\Theta\} \dashrightarrow \mathbf{P}(D_n) \cup \{\Theta\}$$

is as follows:

$$f:D_0 \cup \{\Theta\} \cup \{\delta\} \times \ldots\ldots \times D_{n-1} \cup \{\Theta\} \cup \{\delta\} \dashrightarrow \mathbf{P}(D_n) \cup \{\Theta\} \cup \triangle .$$

With this extension, however, any evaluation of f will return either Θ, $\triangle$ or a set of values (excluding Θ and δ). Sets which are only partially unknown will not be allowed. We choose, therefore, the following extension instead:

$$f : D_0 \cup \{\theta\} \cup \{\delta\} \times \ldots . \times D_{n-1} \cup \{\theta\} \cup \{\delta\} \dashrightarrow \mathbf{P}(D_n \cup \{\delta\}) \cup \{\theta\} \cup \triangle \, .$$

Partially unknown sets are now possible. In fact, the following three types of set (involving unknown elements) are possible:

- a set which is completely unknown , i.e. $\triangle$
- a set which has some unknown elements , i.e. the number of unknown elements is known
- a set which may have some other elements , i.e. the number of unknown elements is not known.

We will not distinguish between the second and the third class because our aim is to make use of all available information rather than discovering the degree of incompleteness. This simplification will reduce the formalism considerably.

Thus, in general, the extended form of the set-valued function f above, when given the arguments $x_0, x_1, \ldots .$ and x_{n-1} , may return either one of:

i) $f(x_0, x_1, \ldots ., x_{n-1})$

ii) $\triangle$

iii) $A \cup \triangle$ where $A \subseteq f(x_0, x_1, \ldots .., x_{n-1})$

iv) θ .

Extension of the operators

In this section we specify the way in which the operators behave when there are missing values in the database. Precise rules are given for each operator individually.

SET-THEORETICAL OPERATORS

isin

The corresponding operator is $\in$ which is extended to be

$\hat{\in} : \alpha \cup\{\theta\}\cup\{\delta\} \times \mathbf{P}(\alpha \cup\{\delta\})\cup\{\theta\}\cup\triangle \longrightarrow \text{bool} \cup\{\theta\}\cup\{\delta\}$

where $x\hat{\in}A$ is defined as follows:

- if x is neither θ nor δ then
    - if A is neither θ nor includes missing values then $x\hat{\in}A$ is $x\in A$
    - if A is of the form $A'\cup\triangle$ then
        - if $x\in A'$ then $x\hat{\in}A$ is true
        - if $x\notin A'$ then $x\hat{\in}A$ is δ
    - if A is $\triangle$ then $x\hat{\in}A$ is δ
    - if A is θ then $x\hat{\in}A$ is θ
- if x is δ then
    - if A is the empty set then $x\hat{\in}A$ is false
    - if A is neither the empty set nor θ then $x\hat{\in}A$ is δ
    - if A is θ then $x\hat{\in}A$ is θ
- if x is θ then $x\hat{\in}A$ is θ.

This specification can be summarized in the table below:

x isin A

x \ A	$\emptyset$	A'	A'U$\triangle$	$\triangle$	θ
a	F	F or T	T or δ	δ	θ
δ	F	δ	δ	δ	θ
θ	θ	θ	θ	θ	θ

where a is not θ or δ, and A' is a non-empty set.

Whenever appropriate, the specification for operators will be given in the form of tables.

<u>is</u>-<u>subset</u>-<u>of</u>

This operation symbol now corresponds to

$\hat{\subseteq}$: $\mathbf{P}(\alpha\ U\{\delta\})U\{\theta\}U\triangle$ x $\mathbf{P}(\alpha\ U\{\delta\})U\{\theta\}U\triangle$ --> bool $U\{\delta\}U\{\theta\}$

where A $\hat{\subseteq}$ B is determined by the table presented in the next page.

<u>is</u>-<u>true</u>-<u>subset</u>-<u>of</u>

The corresponding operator, $\subset$, is extended in a similar way.

$A \hat{\subseteq} B$

A \ B	$\emptyset$	B'	B'U$\triangle$	$\triangle$	θ
$\emptyset$	T	T	T	T	θ
A'	F	T or F	T or δ	δ	θ
A'U$\triangle$	F	δ	δ	δ	θ
$\triangle$	δ	δ	δ	δ	θ
θ	θ	θ	θ	θ	θ

(A' and B' are not empty.)

<u>union</u>

The operator associated with 'union' is

$\hat{\mathrm{U}} : \mathbf{P}(\alpha\mathrm{U}\{\delta\})\mathrm{U}\{\theta\}\mathrm{U}\triangle \times \mathbf{P}(\alpha\mathrm{U}\{\delta\})\mathrm{U}\{\theta\}\mathrm{U}\triangle \dashrightarrow \mathbf{P}(\alpha\mathrm{U}\{\delta\})\mathrm{U}\{\theta\}\mathrm{U}\triangle$

where $A\hat{\mathrm{U}}B$ is defined by the following table:

A \ B	B'	B'U $\triangle$	$\triangle$	θ
A'	A'UB'	A'UB'U$\triangle$	A'U $\triangle$	θ
A'U$\triangle$	A'UB'U$\triangle$	A'UB'U $\triangle$	A'U $\triangle$	θ
$\triangle$	B'U $\triangle$	B'U $\triangle$	$\triangle$	θ
θ	θ	θ	θ	θ

intersection

`intersection` is associated with

$\hat{\cap} : \mathbf{P}(\alpha \cup \{\delta\}) \cup \{\theta\} \cup \Delta \times \mathbf{P}(\alpha \cup \{\delta\}) \cup \{\theta\} \cup \Delta \dashrightarrow \mathbf{P}(\alpha \cup \{\delta\}) \cup \{\theta\} \cup \Delta$

where $A \hat{\cap} B$ is given by

A \ B	B'	B'UΔ	Δ	θ
A'	$A' \cap B'$	$(A' \cap B') \cup \Delta$	Δ	θ
A'UΔ	$(A' \cap B') \cup \Delta$	$(A' \cap B') \cup \Delta$	Δ	θ
Δ	Δ	Δ	Δ	θ
θ	θ	θ	θ	θ

(As before, A' and B' are arbitrary sets which do not contain unknowns.)

Union

corresponds to $\hat{\bigcup} : \mathbf{P}(\mathbf{P}(\alpha \cup \{\delta\})) \cup \{\theta\} \cup \Delta \dashrightarrow \mathbf{P}(\alpha \cup \{\delta\}) \cup \{\theta\} \cup \Delta$

where $\hat{\bigcup} W$ is defined as follows:

- if $\forall X \in W$ X does not include unknowns then

  $\hat{\bigcup} W$ is $\{x \mid x \in X \ \&\ X \in W\}$

- if $\exists X \in W$ X may include unknowns then

  $\hat{\bigcup} W$ is $\{x \mid x \in X' \ \&\ (\exists X \in W : X' \subseteq X)\} \cup \Delta$

- if W is Δ then $\hat{\bigcup} W$ is Δ
- if W is θ then $\hat{\bigcup} W$ is θ.

Intersection

corresponds to $\hat{\cap}$: $\mathbf{P}(\mathbf{P}(\alpha \cup\{\delta\}))\cup\{\theta\}$ --> $\mathbf{P}(\alpha \cup\{\delta\})\cup\{\theta\}$

where $\hat{\cap}$W is:

- if $\forall X \in W$ X does not include any unknown then

  $\hat{\cap}$W is $\{x \mid \forall X \in W\ x \in X\}$

- if $\exists X \in W$ X may include unknown then

  $\hat{\cap}$W is $\{x \mid \forall X \in W\ x \in X\} \cup \triangle$

- if W is $\triangle$ then $\hat{\cap}$W is $\triangle$
- if W is θ then $\hat{\cap}$W is θ.

ARITHMETIC OPERATORS

+ now associates with the operator

$\hat{+}$: int $\cup\{\delta\}\cup\{\theta\}$ x int $\cup\{\delta\}\cup\{\theta\}$ --> int $\cup\{\delta\}\cup\{\theta\}$

where

$$a \hat{+} b = \begin{cases} a + b & \text{if a and b are integers} \\ \delta & \text{if at least one of a and b is } \delta\text{, and neither of them is } \theta \\ \theta & \text{otherwise} \end{cases}$$

The operators corresponding to - , * and / are extended in a similar way.

COMPARISON OPERATORS

is

is now associated with

$\hat{=}$: $\alpha \cup\{\delta\}\cup\{\theta\}$ x $\alpha \cup\{\delta\}\cup\{\theta\}$ --> bool $\cup\{\delta\}\cup\{\theta\}$

where

$$a \mathrel{\hat{=}} b \text{ is } \begin{cases} \text{bool} & \text{if neither a nor b is } \delta \text{ or } \Theta, \\ & \quad\text{(or if a and b are sets, they do not contain } \delta) \\ \delta & \text{if neither a nor b is } \Theta, \text{ and if at least one of them is } \delta \\ & \quad\text{(or, if sets, at least one of a and b contains } \delta) \\ \Theta & \text{otherwise} \end{cases}$$

The operation of "is-not" can be extended correspondingly.

<u>GT</u>

corresponds to the operation:

$\hat{>}$: int U{δ}U{Θ} x int U{δ}U{Θ} --> bool U{δ}U{Θ}

where

$$a \mathrel{\hat{>}} b = \begin{cases} \text{bool} & \text{if a and b are integers} \\ \delta & \text{if neither a nor b is } \Theta, \text{ and at least one of them is } \delta \\ \Theta & \text{otherwise} \end{cases}$$

Operations corresponding to <u>GE</u> , <u>LT</u> and <u>LE</u> are extended in a similar way to that of <u>GT</u>.

LOGICAL CONNECTIVES

<u>and</u>

This symbol stands for the function

$\hat{\&}$: bool U{δ}U{Θ} x bool U{δ}U{Θ} --> bool U{δ}U{Θ}

where $a \mathbin{\hat{\&}} b$ is determined by the table given in the next page.

a \ b	T	F	δ	θ
T	T	F	δ	θ
F	F	F	F	F
δ	δ	F	δ	δ
θ	θ	F	δ	θ

<u>or</u>

The corresponding operation is extended to be

$\hat{v}$: bool U{δ}U{θ} x bool U{δ}U{θ} --> bool U{δ}U{θ}

where $a\hat{v}b$ is determined by the table below:

a \ b	T	F	δ	θ
T	T	T	T	T
F	T	F	δ	θ
δ	T	δ	δ	δ
θ	T	θ	δ	θ

not

\`not\` has the following function associated with it:

$\hat{\neg} : bool\ U\{\delta\}U\{\theta\} \dashrightarrow bool\ U\{\delta\}U\{\theta\}$

where

$$\hat{\neg}\, a = \begin{cases} bool & \text{if a is a boolean value} \\ \delta & \text{if a is } \delta \\ \theta & \text{otherwise} \end{cases}$$

implies

corresponds to the function

$\hat{\rightarrow} : bool\ U\{\delta\}U\{\theta\} \times bool\ U\{\delta\}U\{\theta\} \dashrightarrow bool\ U\{\delta\}U\{\theta\}$

and $a \hat{\rightarrow} b \equiv \hat{\neg} a \hat{\vee} b$.

MISCELLANEOUS OPERATORS

The operators corresponding to Sum, Prod, Average, Max and Min can be extended in two alternative ways. They may be extended to either

- avoid computation and return the value δ whenever the given multi-set contains δ, or
- carry out computation based on the values existing in the given multi-set, and whenever the multi-set contains δ indicate that the result value is not the true result.

However, only the first method maintains our essential principle. Recall that should the existing δ's be replaced by values other than δ, the value of any query must only be closer to reality and it must not change. For example, if the answer to the query "youngest employee in the Sales

Department" when unknown values exist somewhere in the database is 30, then if the unknowns are replaced by other values, it must not change to another value, say 28.

The second method of extension does not agree with this principle. We therefore take the first alternative for the extensions.

Sum

The operator corresponding to

$\hat{\Sigma}$: $\mathbf{M}$(int U{δ})U{θ} --> int U{δ}U{θ}

where

$$\hat{\Sigma}(A) = \begin{cases} \text{an integer} & \text{if } A \text{ is not } \theta \text{ and if it does not contain } \delta \\ \delta & \text{if } A \text{ is not } \theta \text{ and if it contains } \delta \\ \theta & \text{otherwise.} \end{cases}$$

The extension of the operators associated with "Prod", "Average", "Max" and "Min" is similar to the above.

No-of

The corresponding operator is

$\hat{\#}$: $\mathbf{M}$(αU{δ})U{θ} --> int U{δ}U{θ}

where

$$\hat{\#}(A) = \begin{cases} \text{an integer} & \text{if } A \text{ is not } \theta \text{ and if it does not contain } \delta \\ \delta & \text{if } A \text{ is not } \theta \text{ and if it contains } \delta \\ \theta & \text{otherwise.} \end{cases}$$

The Values of Queries

Evaluation of queries in the presence of missing values is carried out in the way specified in section B of chapter III and according to the extensions defined in this chapter. We do not repeat the lengthy specification, and only state the necessary modifications.

Evaluation of expressions involving quantifiers "forall" and "exist" are carried out in accordance with the following extensions:

forall

The value of the expression forall X:e is

- true   if e is true for all values of X
- false   if e is false for some values of X
- δ if e is either true or δ for all values of X but δ for some
- θ otherwise.

(Recall that the variables may not take the values δ or θ.)

exists

An expression of the form exists X:e has the value

- true   if e is true for some value of X
- false   if e is false for all values of X
- δ if e is either false or δ for all values of X but δ for some
- θ otherwise.

Evaluation of expressions of the form $\{ e \}X$ does not change. However, a minor alteration is needed for the evaluation of the expressions of the form $\{ e_0 \mid e_1 \}X$.
Evaluation of $\{e_0 \mid e_1 \}X$ is carried out as before except that if e_1 takes the value δ at least for one value of X then the result value is the union of the set constructed by the normal set constructor and $\triangle$. (The alternative method is to add a δ to the solution set whenever e_1 evaluates to δ. This method, however, does not hold our principle of growing certainty because, for example, the set $\{\delta\}$ after adding information to the database may change to the empty set.)

<u>Examples</u>

Consider the warehouse database from chapter III again. We assume that only part of the information presented in figure 3 is now available. For example, let us assume that, among others, the locations of some of the parts are not known and that we do not know the weight of some of the parts. The available information is presented in figure 4.

We now evaluate some of the queries formulated in chapter III again.

1 . What is the name of the part stored in location 2_1 ?

name-of(item-of(2_1))

The database does not know which item is stored in location 2_1 (i.e. item-of(2_1) is δ). The function name-of applied to δ results in δ, thus the answer to this query is δ.

2 . All parts which are heavier than 200 grams.

{ P | weight-of(P) GT 200 }P

The set {z_13}U △ is the answer to this query since the weight of z_13 is 250 and the weights of pt_12 and it_4 are unknown. Compare this result with the set {z_13 , it_4} which is the answer we obtained from the complete database in chapter III.

3 . Locations of those parts which can substitute mk_3200.

{ places-of(PART) | PART isin substitutes-of(mk_3200) }PART

The parts substituting mk_3200 are pt_10 and pt_12. The location of pt_10 is known and it is 3_1. However, information on the locations of pt_12 is incomplete . Thus the set { {3_1} , {δ , 1_1} } is returned.

4 . How many locations are occupied by the substitutes of the part mk_3200? (i.e. Count the locations appearing in the result of the previous query.)

No-of(Union { places-of(P) | P isin substitutes-of(mk_3200)}P)

Since an unknown value exists in the operand of 'No-of' the answer to this query is δ.

Figure 4

An incomplete database

(Warehouse database of chapter III)

places-of

$f_0: p^+ \dashrightarrow \mathbf{P}(\lambda)^+$

{ <pt_12 , {1_1 , δ}> , <it_4 , { δ }> , <mk_3200 , {1_2}> ,
<z_13 , {δ , 2_4}> , <pt_10 , {3_1}> , <z_100 , {2_2}> }

item-of

$f_1: \lambda^+ \dashrightarrow p^+$

{ <1_1 , pt_12> , <1_2 , mk_3200> , <2_2 , z_100> , <2_4 , z_13> ,
<3_1 , pt_10> }

is-containing

$f_2: \lambda^+ \times p^+ \dashrightarrow B^+$

This function has the same information as the two above.

name-of

$f_3: p^+ \dashrightarrow s^+$

{ <pt_12 , "nut"> , <it_4 , "bolt"> , <mk_3200 , "nut"> ,
<z_13 , "washer"> , <pt_10 , "nut"> , <z_13 , "pipe"> }

substitutes-of

$f_4: p^+ \dashrightarrow \mathbf{P}(p)^+$

{ <pt_12 , Ø > , <it_4 , Ø > , <mk_3200 , { δ , pt_10}> ,
<z_13 , Ø > , <pt_10 , {pt_12}> , <z_10 , Ø > }

Figure 4 (continued)

<u>weight-of</u>

$$f_5: p^+ \dashrightarrow \mathbb{Z}^+$$

{ <pt_12 , 6> , <it_4 , 6> , <mk_3200 , 150> , <z_13 , 250> ,
<pt_10 , 100> , <z_100 , 20> }

<u>qty-of</u>

$$f_6: p^+ \times \lambda^+ \dashrightarrow \mathbb{Z}^+$$

{ <<pt_12 , 1_1> , 10> , <<mk_3200 , 1_2> , 6> , <<z_13 , 2_4> , 10>
<<pt_10 , 3_1> , 60> , <<z_100 , 2_2> , 30> }

<u>capacity-of</u>

$$f_7: \lambda^+ \dashrightarrow \mathbb{Z}^+$$

{ <1_1 , 10> , <1_2 , 60> , <1_3 , 20> , <2_1 , 60> , <2_2 , 40> ,
<2_3 , 50> , <2_4 , 60> , <3_1 , 70> , <3_2 , 5> }

E. Some Remarks

There are other evaluation rules for queries that have been suggested in the literature.However, they do not preserve our main principle stated in section C of this chapter. We pointed out one example when discussing extension of the "Sum" operator.

A great deal of study has been focused on various ways of extending the logical connectives, in particular, implication. Preserving more of the tautologies of the classical two-valued logic (C2) has been the main motivation for these studies. Although all formulae which are tautologies in a 3-valued logic system are also tautologies in C2, it is not the case that C2 tautologies will hold in systems with more truth values. A simple example is $P \rightarrow P$ which fails to be a tautology in most many-valued systems (including ours). Those who have maintained this particular case, have failed to cope with some other obvious ones, such as $\neg(\alpha \,\&\neg\, \alpha)$, because the relation $P \rightarrow Q \equiv \neg P \vee Q$ does not hold in their systems.

Therefore, we have not put any emphasis on maintaining tautologies.

Exhaustive testing has been suggested as a method for obtaining better answers from an incomplete database. For example, consider function f on D_0, D_1, ... and D_{n-1} and its extension f^+. Given the arguments $d_0 \in D_0$, $d_1 \in D_1, \ldots,$ $d_{i-1} \in D_{i-1}$, $d_{i+1} \in D_{i+1}, \ldots$ and $d_{n-1} \in D_{n-1}$, if $\forall a \in D_i$ $f^+(d_0, \ldots, a, \ldots, d_{n-1})$ is b , then the value b is also assumed for $f^{+}(d_0, \ldots, \delta, \ldots, d_{n-1})$. Based on this technique, in [Va-79] an economical algorithm is presented for the evaluation of queries. It assigns, for example, the value true to $P \vee \neg P$ after testing all possible values for P and ensuring that the formula cannot be false or unknown (e.g. the value true can be assigned to

age-of(Jack)$\leq$50 or age-of(Jack)>50

even if age-of(Jack) is unknown). This method can be used when the number of tests is not large, but, it becomes impossible when a quantifier exists in the expression.

In a number of cases we can extract more information from the system. These cases are, again, within the truth tables for the logical connectives. For example, with our evaluation rules for conjunction, $P \mathbin{\hat{\&}} Q$ takes the value δ when P is θ and Q is δ (or vice-versa). We note that $\theta \mathbin{\hat{\&}} \delta$ (or $\delta \mathbin{\hat{\&}} \theta$) cannot possibly be true, because if δ is replaced by the value true then $\theta \mathbin{\hat{\&}} T$ is θ, and if, on the other hand, δ is changed to false then $\theta \mathbin{\hat{\&}} F$ is F. Similarly, $\theta \mathbin{\hat{\vee}} \delta$ can only be either T or θ (the value false is not possible for $\theta \mathbin{\hat{\vee}} \delta$). We avoid adding cases to the truth tables because the extra information cannot play a major role in the evaluation of queries.

There are also alternative methods of defining how θ behaves, but we find them invalid. For instance, when specifying the rules for evaluation of "isin" we assigned θ to $x \mathbin{\hat{\in}} A$ when x is θ and A is the empty set. Some may argue that nothing can belong to the empty set and $\Theta \hat{\in} \emptyset$ must result in false. However, we find the result "false" misleading and thus inappropriate. Suppose that Jack has no sisters and there are no students registered for the course EE2. If a query asks whether Jack's sister is registered for EE2 or not , then "false" is not the correct answer because it is misleading.

Chapter VI

Chapter VI

VARQA VIEWED IN TERMS OF RELATIONAL ALGEBRA

Every application-oriented abstract formalism must be implementable. Only in this way can existing software methods be tested against mathematical standards. A system which is defined in an abstract manner (i.e. its specification given independently from representation and implementation techniques) is eventually confronted with the question of "feasibility of implementation".

We show informally in this chapter that our proposed language does not comprise concepts which are unusually powerful, and it can in principle be implemented.

Varqa is a language based on expressions ; every expression consists of an operator applied to other expressions or to variables denoting the data objects in the database. Many of the operators of Varqa are common in programming languages ; the possibility of their implementation, therefore, need not be shown. Simple operators such as arithmetic operators (e.g. +, *) and set-theoretic operators (e.g. $\cup, \cap$) fall into this category. However, there are other operators in our language whose implementation is a non-trivial task; among them are the variable binding operators.

We will therefore consider a simple functional language which includes all those features of Varga whose implementation is more demanding. We then show that for any expression in this language there exists an equivalent expression in a language based on relational algebra.

In the following section, we will define an algebra of relations which is well suited for this correspondence. The definition of this algebra is mainly based on the work presented in [HHT-75]. The simple functional query language is then mapped to a language based on this algebra.

This work is in a way an informal extension of the work on "cylindric algebras" reported in [HMT-71]. Cylindric algebra is the extended form of Boolean algebra specifically designed as an algebraic apparatus for studying first order predicate logic. The motivation for the introduction of cylindric algebra was to answer, in an algebraic way, metalogical questions of the kind "given a fixed set of sentences $\mathbf{S}$, is a given sentence s implied by the set $\mathbf{S}$?". It was proved [HMT-71] that each such problem can be reduced to an algebraic problem of the form "does a particular formula (associated with s above) hold in the algebra $\mathbf{A}_{\mathbf{S}}$?".

Cylindrical algebra is concerned with expressions of first order logic. Our language is richer than first order predicate logic because it includes the set construction operator.

Our intention in this chapter is not to propose a method of implementation, rather we intend merely to point out that our language is not incompatible with the current software methods.

A. A Simple Functional Query Language (SFQL)

In this section we define a simple functional query language (SFQL) which is based on the principles of Varqa but is considerably simpler . SFQL does not contain all of the operators of Varqa; those operators which are common in programming languages (and the possibility for their implementation need not be shown) have been left out.

The operators allowed in SFQL are the boolean operators ("GT", "LT",....) , the logical connectives ("and" , "or" and "not"), quantifiers ("exists" and "forall") and the set construction operator. Although we exclude many of the operators existing in Varqa, SFQL is still a powerful query language. For example, using these operators only, we can still formulate the query "all employees who earn more than their managers".

We put a restriction on the expressions of SFQL by not permitting nested use of the set construction operator. (With this restriction we simplify the translation considerably.) The results obtained here can later be generalised to allow nested set construction.

With the exception of the restriction on nested set construction, the formation rules for the expressions of SFQL are the same as those of Varqa.

B. Relational Algebra, an alternative approach

Relations can be viewed as tables with rows and columns [Ul-80]. The order of rows is unimportant, but in the definition of some relational algebra operators reference has been made to the ordering of columns. Many database researchers, e.g. Codd [Co-79], claim that, by referring to a column of a relation by a name instead of its position, the ordering becomes unimportant. However, the transformation is not straightforward and causes problems. For example, Codd's definition for the operator projection (which makes reference to the order of columns) is not adequate when the columns of relations are identified by names. Consider a relation with two columns identified by "A" and "B". If we use the projection operator to construct a two column relation with both columns being the original A-column, then the names cannot simply be inherited [HHT-75].

In the context of our work, it is easier to map the expressions of SFQL to a language based on an algebra of relations which is independent of column ordering (because in SFQL we make use of names and not ordering). Therefore we define relations with "column name" rather than "column ordering". We refer to the column names as "selector names" (i.e. selector names are names for the "underlying domains" of the relations). We assume a universal set ω which contains all the underlying domains.

Given a set of selector names S, <u>a tuple with selector set S</u> is a function which assigns a value (from ω) to each selector name of S. For each selector name s in S, t(s) is called "the value of s for the tuple t". For any selector set S, W(S) is the set of all tuples with selectors S.

Given a set of selector names S, <u>a relation with selector set S</u> is the set S together with a set of tuples with selector set S. The <u>arity</u> of a <u>relation with selector set S</u> is the number of elements of S.

[Recall from chapter I (when we defined a relation as a cartesian product of sets) we argued against the definition of arity for relations defined in that manner. Defining "arity of relations with selectors" is not contradicting our objection to "arity of relations". Note that for each set S of selector names we have a distinct empty relation Ø(S).]

Example: Consider a relation R represented in the form of a table:

PART-NO	LOCATION	QTY
100	90	100
150	23	100
155	75	50
110	95	50

The set of selectors of R is {PART NO , LOCATION , QTY} and the arity of R is 3. A tuple t in R can be represented as

PART NO	LOCATION	QTY
150	23	100

For this tuple the values for the selector names are t(PART-NO)=150 , t(LOCATION)=23 and t(QTY)=100.
Note that the order of the columns is in fact unimportant. Therefore, it is meaningless to talk about Nth column of a relation. We can only refer to a column by its selector name.

We now define a collection of operators which can operate on relations with selector sets [HHT-75].

For any relation R with selector set S, if S' is a subset of S then the projection of R on S' , R % S' is

$$\{ t' \mid \exists t \in R : t' = t!S' \} .$$

(Recall that f!A means function f restricted to the subset A of its domain.) Similar to the projection operator of the ordinary relational algebra, this operator simply removes some of the columns of R (those which are not in S', of course). If we want to duplicate some of the columns then this operator cannot be used because we need different names for the duplicate columns. Thus we define another operator which is a generalised form of projection (We simply call it "generalised projection".).

A "renaming function" in this context is a function which maps a set of selector names to another one.

Given a relation R with selector set S, and a renaming function γ which maps selector set S' to selector set S , the <u>generalised projection</u> of R over γ, denoted by $R \Pi \gamma$ is

$$\{ t' \mid t' \in W(S') \;\&\; (\exists t \in R : t' = t \circ \gamma) \}$$

(Recall that $f \circ g$ stands for the composition of functions g and f.) By defining an appropriate γ we can duplicate, rename or indeed remove some of the columns of R.

In ordinary relational algebra, the "union", "intersection" and "set difference" operators require their operands to be compatible; i.e. the relations given to these operators as the arguments must have the same types. Here we do not enforce this restriction on the operands and allow the two argument relations to be of any type.

Given two relations R1 and R2 with selector sets S1 and S2 respectively, then the <u>generalised intersection</u> of R1 and R2, denoted by $R1 \underline{\cap} R2$ is

$$\{ t \mid t \in W(S1 \cup S2) \;\&\; (t!S1) \in R1 \;\&\; (t!S2) \in R2 \}$$

There are two extreme cases in the application of generalised intersection. When the selector sets S1 and S2 are the same (i.e. R1 and R2 are compatible relations) generalised intersection degenerates to set intersection. The other extreme case is when S1 and S2 are completely distinct: generalised intersection then returns the concatenation (in relational database jargon: cartesian product) of

R1 and R2. In other cases, when S1 and S2 overlap, the equijoin of R1 and R2 on the intersection of S1 and S2 is the result.

Similarly, generalised union of relations R1 with selector set S1 and R2 with selector set S2, denoted by $R1 \mathbf{Y} R2$ is defined to be

$$\{ t \mid t \in W(S1 \cup S2) \;\&\; ((t!S1) \in R1 \vee (t!S2) \in R2) \}$$

Generalised union will also degenerate to the union operator of the ordinary relational algebra when S1=S2.

Generalised difference is also similar. For any two relations R1 and R2 with selector sets S1 and S2 respectively the generalised difference of R1 and R2 , R1-R2 , is

$$\{ t \mid t \in R1 \;\&\; (t!(S1 \cap S2)) \notin (R2\%(S1 \cap S2)) \}$$

Again, when S1=S2, generalised difference acts as the ordinary operator difference. When S1 and S2 overlap, however, this operator selects from R1 those tuples for which the values corresponding to the common selector names cannot be found in the compatible columns of any tuple in R2.

The notion of selection is covered by generalised intersection. To select (from a relation R) tuples which satisfy certain criteria on some selector names, we construct a relation whose selector set contains only those selector names to which the criteria is applicable and whose tuples are just those which satisfy the criteria (call it relation F). The selection of R over the criteria is the result of generalised intersection between R and F.

Another useful operator is the "complement" operator. The <u>complement</u> of the relation R with selector set S is

$$C(R) = W(S)-R$$

where as before W(S) is the set of all tuples with selector set S.

Relations (with selectors) together with the above operators form a powerful algebraic structure. By applying the operators to relations we can construct expressions. Thus, each expression defines a relation. We call this language Relational Algebra Query Language, and refer to it as RAQL for short.
It is assumed that the reader is familiar with the languages based on relational algebra. Material on this topic is presented in [Me-78].

C. Translation from SFQL to RAQL

We explore in this section, by illustration rather than by fully formal treatment, the correspondence between SFQL and RAQL.

Sets and functions of SFQL are simply regarded as relations. For any set with the type symbol α in SFQL, the equivalent relation in RAQL is a unary relation with an appropriate selector set $\{S_0\}$ which contains all objects of type α.

For any function symbol φ associated with the function f and the type expression $\alpha_0, \alpha_1, \ldots, \alpha_{n-1} \text{----}> \alpha_n$ in SFQL, there exists an equivalent relation in RAQL which has the selector set

$$\{S_0, S_1, \ldots, S_{n-1}, S_n\}$$

and whose tuples are

$$\{\ t \mid t\epsilon W(\{S_0, \ldots, S_n\})\ \&\ f(t!\{S_0\}, t!\{S_1\}, \ldots, t!\{S_{n-1}\}) = t!\{S_n\}\}$$

We start with the simplest expressions of SFQL and show informally that for every expression in SFQL there exists an equivalent expression in RAQL.

If φ is a function symbol with the type expression $\alpha_0, \alpha_1, \ldots, \alpha_{n-1} \text{----}> \alpha_n$ in SFQL which corresponds to the relation R_φ with selector set $\{S_0, S_1, \ldots, S_n\}$ in RAQL then

$\varphi(a_0,a_1,\ldots,a_{n-1})$ (where a_i is of type α_i for all $i \in n$) in SFQL is equivalent to

$(R_\varphi \Omega F) \% \{S_n\}$

where F is a relation with selector set $\{S_0,S_1,\ldots,S_{n-1}\}$ whose only tuple is $\{<S_0,a_0>,<S_1,a_1>,\ldots.,<S_{n-1},a_{n-1}>\}$

in RAQL.

In the above relational expression, $R_\varphi \Omega F$ can be thought of as selection; using the filter relation F we select a row from R_φ. The selected tuple is then projected on S_n to yield the result.

Example:

Consider function symbol grade-of which corresponds to a relation R with selector set { STUDENT, COURSE , GRADE } illustrated as a table below

STUDENT	COURSE	GRADE
John	EE2	23
Fred	FG1	25
Pam	FG3	19

In SFQL, a query which asks for the grade of John in EE2 is formulated as

grade-of(John,EE2)

The filter relation F for this expression is a relation with selector set {STUDENT , COURSE} as shown below

STUDENT	COURSE
John	EE2

$(R \,\underline{\Omega}\, F)$ returns the tuple {<STUDENT,John> , <COURSE,EE2> , <GRADE,23>}. The final result 23 is obtained when this relation is projected on GRADE.

Given n expressions $e_0, e_1, \ldots, e_{n-1}$ corresponding to relations R_{e_0}, $R_{e_1}, \ldots, R_{e_{n-1}}$ respectively, and an n-ary function symbol φ corresponding to relation R_φ with selector set $\{S_0, S_1, \ldots, S_{n-1}, S_n\}$ (S_n corresponds to the range of the function associated with φ), then the RAQL relational expression corresponding to $\varphi(e_0, e_1, \ldots, e_{n-1})$ is

$$((R_{e_0} \,\underline{\Omega}\, R_{e_1} \,\underline{\Omega}\, \ldots \,\underline{\Omega}\, R_{e_{n-1}}) \,\underline{\Omega}\, R_\varphi) \,\%\, \{S_n\}$$

<u>Logical connectives</u>

If the expressions e_1 and e_2 in SFQL correspond to relations L_1 and L_2 in RAQL respectively, then

<u>e_1 and e_2</u>	in SFQL corresponds to	$L_1 \,\underline{\Omega}\, L_2$	in RAQL
<u>e_1 or e_2</u>	in SFQL corresponds to	$L_1 \,\mathbf{Y}\, L_2$	in RAQL
<u>not e_1</u>	in SFQL corresponds to	$C(L_1)$	in RAQL.

Example:

If the corresponding relation to e_1 is R_{e_1} with selector set $\{S_{1_0}, S_{1_1}, \ldots, S_{1_{n1}}\}$ and the corresponding relation to e_2 is R_{e_2} with selector set $\{S_{2_0}, S_{2_1}, \ldots, S_{2_{n2}}\}$ then the expression <u>e_1 and e_2</u> of SFQL is equivalent to the relation

$$\{ t \mid t \in W(\{S_{1_0}, \ldots, S_{1_{n1}}\} \cup \{S_{2_0}, \ldots, S_{2_{n2}}\}) \;\&\; (t!\{S_{1_0}, \ldots, S_{1_{n1}}\}) \in R_{e_1} \;\&\; (t!\{S_{2_0}, \ldots, S_{2_{n2}}) \in R_{e_2}\}.$$

Example:

to illustrate the correctness of the above translation algorithm we show the equivalence of $\neg(\neg P \& \neg Q) \equiv P \vee Q$.

Let e1 and e2 correspond to relation L1 with selector set S1 and relation L2 with selector set S2 respectively. We start with the left hand side:

<u>not(not e1 and not e2)</u> ==>

$$
\begin{aligned}
& C(C(L1) \Cap C(L2)) \\
&= C(W(S1)-L1) \Cap (W(S2)-L2)) \\
&= C(\{t|t\in W(S1) \ \& \ t\notin L1\} \Cap \{t|t\in W(S2) \ \& \ t\notin L2\} \\
&= C(\{t|t\in W(S1 \cup S2) \ \& \ ((t!S1)\in\{r|r\in W(S1) \ \& \ r\notin L1\}) \\
&\qquad\qquad \& \ ((t!S2)\in\{r|r\in W(S2) \ \& \ r\notin L2\})\}) \\
&= C(\{t|t\in W(S1 \cup S2) \ \& \ (t!S1)\notin L1 \ \& \ (t!S2)\notin L2 \ \}) \\
&= W(S1 \cup S2) - \{t|t\in W(S1 \cup S2) \ \& \ (t!S1)\notin L1 \ \& \ (t!S2)\notin L2\} \\
&= \{t|t\in W(S1 \cup S2) \ \& \ \neg((t!S1)\notin L1 \ \& \ (t!S2)\notin L2)\} \\
&= \{t|t\in W(S1 \cup S2) \ \& \ ((t!S1)\in L1 \vee (t!S2)\in L2)\} \\
&= L1 \curlyvee L2
\end{aligned}
$$

==> <u>e1 or e2</u> .

The existential quantifier

If a SFQL expression e corresponds to relation R_e with selector set $\{S_0, S_1,\ldots,S_n\}$ then the relation corresponding to exists X: e is

- R_e itself , if X does not occur in e.
- $R_e\%\{S_0,S_1,\ldots,S_{i-1},S_{i+1},\ldots,S_n\}$ if X occurs in e in the place corresponding to S_i for some $i\in(n+1)$.

Expressions with universal quantifiers are transformed into expressions with existential quantifiers only, $\forall x\ P \equiv \neg\exists x\ \neg P$.

The set construction operator

Given two SFQL expressions e_1 and e_2 corresponding to relation R_{e_1} with selector set $\{S_{1_0}, S_{1_1}, \ldots, S_{1_{n1}}\}$ and relation R_{e_2} with selector set $\{S_{2_0}, S_{2_1}, \ldots, S_{2_{n2}}\}$ then the expression $\{\ e_1 \mid e_2\}X$ of SFQL corresponds to

$(R_{e_1}\ \Omega\ R_{e_2}\ \Omega\ R_X)\%\{S_{1_0}, S_{1_1}, \ldots, S_{1_{n1}}\}$
where R_X is a unary relation with selector set $\{S_n\}$, for some n, containing all possible values for X.

Some examples

Let us consider part of the university database, with three functions associated with "name-of" , "is-taking" and "grade-of" . Suppose these functions correspond to relations R1 , R2 and R3 respectively. The relations are presented in the form of tables on the next page.

relation R1

ST-NO	ST-NAME
1	Jack
2	Jane
3	Pam
4	Fred

relation R2

ST-NO	COURSE
1	EE1
1	Ma
2	EE1
3	Ma
3	EE1
4	Ma
4	KS

Relation R3

ST-NO	COURSE	GRADE
1	EE1	12
1	Ma	11
2	EE1	15
3	Ma	10
3	EE1	12
4	Ma	16
4	KS	10

Here we formulate three queries in SFQL and then translate them into RAQL using the translation rules given above.

1 . Grades in EE1 of all students who take EE1.

{ grade-of(X,EE1) | X is-taking EE1 }X

The equivalent relational expression in RAQL is

$((R3 \underline{\cap} F1) \underline{\cap} (R2 \underline{\cap} F2))\%\{GRADE\}$

where the filter relations F1 and F2 are both {{<COURSE , EE1>}}.

In the evaluation of the RAQL expression, first the relations $(R3 \underline{\cap} F1) = K1$ and $(R2 \underline{\cap} F2) = K2$ are computed:

K1

ST-NO	COURSE	GRADE
1	EE1	12
2	EE1	15
3	EE1	12

K2

ST-NO	COURSE
1	EE1
2	EE1
3	EE1

The result of generalised intersection between K1 and K2 is relation K3 which is the same as K1. The answer to the query is obtained by projecting K3 on GRADE, that is the relation K4.

relation K4

GRADE
12
15

2 . Names of the students who take EE1.

{ name-of(X) | X is-taking EE1 }X

The equivalent expression in RAQL is

$(R1 \underline{\cap} (R2 \underline{\cap} F))\%\{ST\text{-}NAME\}$

where F, the filtering relation, is {{<COURSE , EE1>}}.

The above expression can easily be evaluated in a similar way to the previous one. The answer to the query is the relation

ST-NAME
Jack
Jane
Pam

3 . Name of the students who have a grade below 12.

{ name-of(X) | exists Y: grade-of(X,Y) LT 12 }X

We apply the translation rules starting from the innermost (i.e. grade-of(X,Y) LT 12). As before, F is a filter relation; it is a unary relation on GRADE whose elements are less than 12, i.e. 10 and 11.

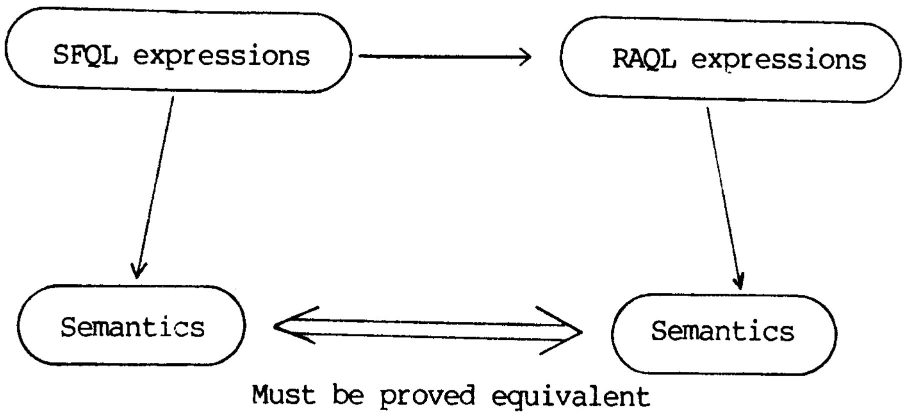
$$(R1 \,\Omega\, (\underbrace{\underbrace{(R3 \,\Omega\, F)}_{\text{grade } <12} \ \% \ \{COURSE\}\ )}_{\text{exists a course}}\ )\ \%\ \{ST\text{-}NAME\}$$

The evaluation of this query will result in the unary relation

ST-NAME
Jack
Pam
Fred

D. Some Remarks

In this chapter we have given an indication of the feasibility of implementation for a powerful subset of Varqa. This work is, of course, not a formal treatment of the topic. For a systematic study, we need to define the two languages formally, then give syntactic transformation rules, and finally prove equivalence for the semantics of corresponding expressions in these two languages . This process is illustrated in the figure below.



This is an interesting topic for research, however, we feel that it is beyond the scope of this thesis. It therefore remains as a suggestion for further work.

Recall that in the introduction of our relational algebra we assumed a universal set (the union of all underlying sets). The universe of the relational algebra corresponding to our SFQL is determined by the types existing in SFQL. We ignored the problem of infinity in the introduction of SFQL. Thus the universe of RAQL can be infinite. It is noted [HHT-75] that when infinite (or, very large) sets are involved then there is a possibility to encounter a number of problems. The most subtle one is that in the computation of some expressions, although the final result is finite, there can be infinite relations required in the intermediate steps. In [HHT-75] a solution is suggested (postulating ordering on sets) to ensure termination for the infinite (or potentially infinite) intermediate computations.

Epilogue

In this thesis we have not only designed a new query language for database systems but have proposed a whole new methodology for designing such languages. The standard design methodology for query languages (and also for all-purpose programming languages) is to start off from operational concepts, then to formulate an appealing syntax, and finally (even then, only in some cases) to state the denotational semantics [Wa-78].

However, our methodology puts great emphasis on the denotational semantics of every aspect of the language. The denotational semantics of Varqa were developed hand in hand with its notation. We extended ordinary algebra to include variable binding operators, and allowed the operators in the algebra to work on the union, cartesian product and the powerset of types.

Varqa lends itself to a number of possible extensions. More operators can be added to the language. Such operators might be: "Top N" (to find the N highest elements of an ordered set), "Bottom N" (to find the N lowest elements for an ordered set) and more variable binding operators such as $\exists!$ (e.g. $\underline{\exists!\ X : P}$, meaning there exists a unique X such that P). The formalism of Varqa can accommodate such operators quite naturally.

This work is by no means complete. As the first follow-up for this work, "updates" must be studied. One important aspect in the study of updates is how to maintain correctness of the system (in database jargon : integrity constraints). Several suggestions have been made on conceptual modelling of updates in database systems. In [DMF-81] every database is considered to be a set of assertions, queries are consequences of these assertions , and updates are modifications to the set of assertions. An alternative approach is to view every database as a set of "database instances" (dbi's) and regard updates as functions mapping one dbi to the next one [Ma-81].

A novel approach to updates is through modal logic where the database instances are viewed as its "possible worlds". (Modal logic, an extension to predicate logic, is the logic of necessity and possibility: a proposition is "necessary" if it holds in all admissible worlds , and it is "possible" if it holds in some worlds [Ch-80] . Modal logic is particularly suited for reasoning about dynamic systems in which time plays a role.) "Integrity constraints" can be naturally viewed as propositions which must be "necessarily" satisfied in every world.

Finally, the object-oriented work on abstract data types can be carried over for studying updates. In fact, a database system can be completely characterised by an algebraic specification of various operations [Gu-77].

On the implementation of Varqa several suggestions prove useful. A method for implementing set processors is proposed in [Ha-76]. The techniques used in the implementation of the programming language SETL [SSS-81] may be applicable for the implementation of Varqa.

Research is in progress for the formal description of arbitrary complex information structures, with the aim of developing a new method for mathematical implementation of database systems [Ma-80].

Lastly, dataflow implementation techniques [Ka-74] may also be applicable to databases. There is yet no evidence to prove that such an approach leads to success. However, there is at least one person who is interested in doing research on it: the author himself!

Bibliography

BIBLIOGRAPHY

[ABU-79] Aho A V , Beeri C , Ullman J D
"The theory of joins in relational databases"
ACM TODS , Vol 4 , No 3 , Sept 1979 , pp 297-314

[Ac-67] Ackerman R
"Introduction to many-valued logics"
Routledge and Kegan Paul 1967

[ANSI-75] ANSI / X3 / SPARC
Study group on data base management systems
"Interim report" FDT (ACM SIGMOD Records) 7,2 1975

[AU-79] Aho A V , Ullman J D
"Universality of data retrieval language"
6th POPL pp 110 - 120
(ACM / SIGPLAN Conf. on Prin. of Prog. Lang. Jan 1979)

[AW-77] Ashcroft E A , Wadge W W
"Lucid, a nonprocedural language with iteration"
CACM Vol 20 , No 7 , July 1977 , pp 519-526

[AW-79] Ashcroft E A , Wadge W W
"A logical programming language"
Waterloo Univ. CS - 79 - 20 , June 79 (revised March 80)

[AW-82] Ashcroft E A , Wadge W W
"Lucid: the dataflow programming language"
Academic Press (To be published)

[Ba-78] Backus J
"Can programming be liberated from the von Neumann style ?
A functional style and its algebra of programs"
CACM , Vol 21 , No 8 , Aug 1978 , pp 613-641

[BB-79] Beeri C , Bernstein P A
"Computational problems related to the design of normal form relational schemes"
ACM TODS , Vol 4 , No 1 , March 1979 , pp 30-59

[BF-79] Buneman P , Frankel R E
"FQL - a functional query language"
Int. Conf. on Management of Data , Boston 1979 , pp 52-58

[Bi-78] Biskup J
"On the complementation rule for multivalued dependencies in database relations"
Acta Informatica Vol 10 , Fasc. 3 , 1978 , pp 297-305

[Bi-81] Biskup J
"A formal approach to null values in data base relations"
in Advances in Database Theory , Vol 1 , (Ed Gallaire, Minker, Nicolas) Plenum Press , 1981 , pp 299-341

[BN-78] Biller H, Neuhold E J
"Semantics of databases : the semantics of data models"
Information Systems Vol 3 , Part 1 , 1978 , pp 11-30

[BNF-81] Buneman P , Nikhil R , Frankel R E
"A practical functional programming system for databases"
Proceedings ACM Conference on Functional Prog. and Machine Architecture , 1981

[CF-58] Curry H B , Feys R
"Combinatory logic"
Vol 1 , North Holland Publishing Company , 1958

[Ch-39] Church A
"On 3-valued logical calculus and its application to the analysis of contradictions"
Journal of Symbolic Logic Vol 4 , 1939 , pp 98-98

[Ch-40] Church A
"A formulation of the simple theory of types"
Journal of Symbolic Logic Vol 5 , 1940 , pp 56-68

[Ch-76] Chen P P S
"The entity-relationship model : Towards a unified view of data"
ACM TODS Vol 1 , No 1 , March 1976 , pp 9-36

[Ch-80] Chellas B F
"Modal logic ; an introduction"
Cambridge University Press , 1980

[Co-70] Codd E F
"A relational model of data for large shared data banks"
CACM Vol 13 , No 6 , June 1970 , pp 371-387

[Co-71] Codd E F
"A database sublanguage founded on the relational Calculus"
ACM SIGFIDET Workshop on Data Description, Access and Control
Proc 1971 pp 35-68

[Co-75] Codd E F
"Understanding relations"
FDT: Bull. ACM/SIGMOD , Vol 7 , Part 3-4 , 1975 , pp 23-28

[Co-79] Codd E F
"Extending the database relational model to capture more meaning"
ACM TODS Vol 4 , No 4 , Dec. 1979 , pp 397-434

[CPP-78] Colombetti M , Paolini P , Pelagatti G
"Nondeterministic languages used for the definition of data models"
in Logic and Databases (Gallaire, Minker eds.)
Plenum Press 1978 pp 237-258

[Da-81] Data C J
"An introduction to database systems"
The systems programming series , Third Edition
Addison Wesley 1981

[DMF-81] Dos Santos C S , Maibaum T S E , Furtado A L
"Conceptual modelling of database operations"
To appear.

[Fa-77] Fagin R
"Multivalued dependencies and a new normal form for relational databases"
ACM TODS Vol 2 , No 3 , Sept. 1977 , pp 262-278

[Fa-81] Fagin R
"A normal form for relational databases that is based on domains and keys"
ACM TODS Vol 6 , No 3 , Sept. 1981 , pp 387-415

[GG-78] Gotlieb C C , Gotlieb L R
"Data types and Structures"
Prentice - Hall 1978

[GH-78] Guttag J V , Horning J J
"The algebraic specification of abstract data types"
Acta Informatica Vol 10 , No 1 , 1978 , pp 27-52

[GM-78] Gallaire H , Minker J
"Logic and Databases"
Plenum Press , New York , 1978

[GMN-81] Gallaire H , Minker J , Nicolas J M
"Advances in database theory"
Vol 1 Plenum Press , New York , 1981

[Gr-77] Grant J
"Null values in a relational database"
Information Processing letters Vol 6, No 5 , Oct 1977 ,
pp 156-157

[GTW-78] Goguen J A , Thatcher J W , Wagner E G
"An initial algebra approach to the specification, correctness and implementation of abstract data types"
in Current Trends in Programming methodology Vol IV ,
Data Structuring (Yeh , ed.) Prentice Hall , 1978 pp 80-149

[Gu-77] Guttag J E
"Abstract data types and the development of data structures"
CACM Vol 20 , No 6 , June 1977

[Ha-76] Hardgrave W T
"A technique for implementing a set processor"
SIGPLAN notices , Vol 11 , Special Issue , pp 84-96 1976
(Proc. Conf. on Data: abstraction, definition and structure)

[HHT-75] Hall P A V , Hitchcock P , Todd S J P
"An algebra of relations for machine computation"
IBM - UK report UKSC 0066 , Jan 1975

[HM-79] Hammer M , McLeod D
"The semantic data model : A modelling mechanism for data base application"
Proc. Int. Conf. on Management of Data (SIGMOD) Austin , 1979
pp 26-36

[HMT-71] Henkin L , Monk J D , Tarski A
"Cylindric algebras"
Part 1 , North-Holland Publishing Co. , 1971

[IL-81] Imielinski T , Lipski W
"On representing incomplete information in a relational database"
Proc. 7th VLDB , Cannes , France , Sept 1981 , pp 388-397

[Ka-74] Kahn G
"The semantics of a simple language for parallel programming"
Proceedings of IFIP Congress (Rosenfeld editor)
pp 471-475 , 1974

[Ke-78] Kent W
"Data and Reality"
North Holland Publishing Company , 1978

[Ke-79] Kent W
"Limitations of record based information modes"
ACM TODS Vol 4 , No 1 , March 1979 , pp 107-131

[Kl-62] Kleene S C
"Introduction to meta-mathematics"
D.Van Nostrand Company 3rd reprint , 1962

[KMM-80] Kalish D , Montague R , Mar G
"Logic , Techniques for Formal Reasoning"
Harcourt Brace Jovanovich, inc.
Second Edition 1980

[Ko-79] Kowalski R
"Logic for problem solving"
Artificial Intelligence Series , North Holland , 1979

[Ko-81] Kowalski R
"Logic as a database language"
Imperial College , London , July 1981

[La-66] Landin P J
"The next 700 programming languages"
CACM Vol 9 , No 3 , March 1966 , pp 157-166

[Li-79] Lipski W
"On semantic issues connected with incomplete information"
ACM TODS Vol 4 , No 3 , Sept 1979 , pp 262-296

[Li-81] Lipski W
"On database with incomplete information"
Journal of ACM Vol 28 , No 1 , Jan 1981 , pp 41-70

[LP-82] Louis G , Pirotte A
"A denotational definition of the semantics of DRC, a Domain Relational Calculus"
(To appear) Proc. 8th VLDB Conference , Mexico , Sept. 1982

[LT-77] Legard H F , Taylor R W
"Two views of data abstraction"
CACM Vol 20 , No 6 , June 1977 pp 382-384

[Ma-74] Manna Z
"Mathematical theory of computation"
McGraw - Hill 1974

[Ma-76] Markowsky G
"Chain - complete posets and directed sets with applications"
Algebra Univ. 6 , Birkhauser Verlog Basel , 1976 , pp 53-68

[Ma-77] Maibaum T S E
"Mathematical semantics and a model for databases"
Proc. IFIP 1977 (Gilchrist ed.) , pp 133-138

[Ma-80] Mandarella R
Private correspondence

[Ma-81] Maibaum T S E
Private communication

[MB-79] Mac Lane S , Birkhoff G
"Algebra"
Macmillan Publishing Co. 2nd Edition , 1979

[Me-53] Menger K
"On variables in mathematics and in natural science"
British Jour. of Phil. Sci. 5 , 1954 , pp 134-142

[Me-78] Merrett T H
"The extended relational algebra , a basis for query languages"
In Databases; Improving usability and responsiveness , (Shneiderman ed.) Academic Press 1978

[Me-79] Mendelson E
"Introduction to Mathematical logic"
D.Van Nostrand Co. N.Y. Second Edition 1979

[Mo-54] Moh Show-kwei
"Logical paradoxes for many-valued systems"
Journal of Symbolic Logic Vol 19 1954 pp 37-40

[Ol-78] Olle T W
"The Codasyl approach to database management"
A Wiley-Interscience publication,
John Wiley & Sons , 1978

[Pa-76] Park D M R
"Finiteness is MU-ineffable"
Theoretical Computer Sci. , Vol 3 , 1976 , pp 173-181

[Pa-82] Park D M R
Private Communication

[Pi-78] Pirotte A
"High level database query language"
in Logic & Databases (Gallaire Minker eds.)
Plenum Press , New York , 1978 , pp 409-436

[Re-69] Rescher N
"Many-valued logic"
McGraw-Hill Inc. , 1969

[Re-76] Rem M
"Associons and the closure statement"
Mathematical Centrum , Amsterdam , 1976

[Ru-03] Russell B
"The principles of Mathematics"
George Allen & Unwin Ltd , Eighth impression , 1964

[Sc-71] Schwartz J T
"Abstract and concrete problems in the theory of files"
in Data Base Systems : Courant Computer Science Symposium 6
May 24-26 , 1971 , Ed. R. Rustin , Prentice Hall

[Sc-76] Scott D
"Data types as lattices"
SIAM Journal on Computing , Vol 5 , 1976 , pp 522-587

[Se-75] Senko M E
"Information systems: records, relations, sets, entities and things"
Information Systems , Vol 1 , No 1 , 1975 pp 1-13

[Sh-81] Shipman D W
"The functional data model and the data language DAPLEX"
ACM TODS , Vol 6 , No 1 , March 1981,
pp 140-173

[SS-77] Smith J M , Smith D C P
"Database abstractions : aggregation and generalization"
ACM TODS , Vol 2 , No 2 , June 1977 , pp 105-135

[SSS-81] Schonberg E , Schwartz J T , Sharir M
"An automatic technique for selection of data representations in SETL programs"
ACM TOPLS Vol 3 No 2 pp 126-143 April 1981

[St-77] Stoy J E
"Denotational semantics : the Scott-Strachey approach to programming language theory"
The MIT Press , 1977

[St-80] Steyer F
"A uniform description of database management systems"
Information Systems , Vol 5 , No 2 , 1980 , pp 127-135

[SY-78] Sagiv Y , Yannakakis M
"Equivalence among relational expressions with union and differences operators"
Proc. VLDB , 1978 pp 535-548

[Ta-55] Tarski A
"A lattice theoretical fixpoint theorem and its applications"
Pacific Journal of Mathematics , Vol 5 , 1955 , pp 285-309

[TWW-78] Thatcher J W , Wagner E G , Wright J B
"Data type specification : parameterization and the power of specification technique"
Proc. SIGACT 10th Annual Symp. on Theory of Comp. pp 119-132

[Tu-81] Turner D A
"Recursion equations as a programming language"
Presented at Newcastle Functional Prog. Workshop , July 1981

[Ul-80] Ullman J D
"Principles of database systems"
Computer Science Press , 1980

[Va-79] Vassiliou Y
"Null values in data base management : a denotational semantics approach"
ACM / SIGMOD International Symp. on Management of Data , 1979 , pp 162-169

[VM-81] van Edam M H , Maibaum T S E
"Equations compared with clauses for specification of abstract data types"
in Advances in database theory, Vol 1 , (Ed Gallaire, Minker, Nicolas) Plenum Press , 1981 , pp 159-194

[Wa-78] Wadge W W
"Away from the operational view of Computer Science"
Theory of Computation Report , University of Warwick
No 26 , November 1978

[We-76] Weber H
"A semantic model of integrity constraints on a relational database"
Modelling in Data Base Management Systems , G.M.Nijssen (ed), North Holland Publishing Co. , 1976

[Zi-75] Zilles S N
"An introduction to Data Algebras"
Working draft paper , IBM Research Laboratory , San Jose , Calif.

[Zl-75] Zloof M M
"Query-by-Example : operations on transitive closure"
IBM Research RC 5526 , July 1975