Fundamenta Informaticae 26 (1996) 31-57
IOS Press

A UNIVERSAL MODEL FOR NON-PROCEDURAL DATABASE LANGUAGES

Naphtali RISHE
High-performance Database Research Center, School of Computer Science, Florida International University, University Park, Miami, FL 33199, USA

Abstract. We propose a language which can express every computable query. The language is syntactically based on the first-order predicate calculus, but semantically is interpreted as post-conditions, unlike the customary calculus query languages. The language has a capability to restrict itself to reasonable queries, accepting criteria of reasonability as a parameter.

1. Introduction

Query languages having the expressibility power of the relational calculus (or, equivalently, relational algebra) are often called *"complete"* (Codd-complete [1]). However, they cannot express many reasonable queries. Many extensions of the calculus were considered in the literature: aggregates, transitive closure, fixed points, Horn clauses (in Prolog-like languages), high-order logic. They are able to specify *more* queries. Yet, no proposed non-procedural query language can express *all* reasonable queries.

This paper proposes a language which can express *every* query, provided the query is potentially computable. (The computability of queries is formally defined by modifying the theory of partial recursive functions to apply to the domain of databases, which *may* be *order-less* finite sets, rather than to the domain of integers.)

The language is syntactically based on the *first-order* predicate calculus, but semantically is interpreted as *post-conditions,* not like the customary calculus query languages. Yet, I believe that the postconditional interpretation has a better appeal to intuition and is more non-procedural than the customary one.

Being able to express every query, the language may allow unreasonable queries that should be prohibited (*e.g.* a query to find the *average* social security number). However, the language has a capability to restrict itself to reasonable queries, accepting criteria of reasonability as a parameter. This allows for screening out *syntactically* those queries which would be unreasonable according to the parametrically given reasonability criteria. (In fact, it is rather a language *model*, than one language. The model produces particular sublanguages according to criteria of reasonability.)

Additional features of this non-procedural language include:

- It allows non-deterministic queries. This exempts the user from the specification of details which the user does not consider important. It also gives a greater potential for optimization. Nevertheless, every query can be specified deterministically, if needed.

- It can express not only queries, but also *all* possible update transactions, integrity constraints, userview definitions, *etc.* in the relational and semantic semantic database models.
- In addition to all of the terminating computable queries, *etc.*, this language can express all of the *partially* computable queries, *etc.*

An instantaneous data base is a finite structure of facts (elementary propositions) which is regarded as describing a state of an application world. A data base schema describes time-independent properties of an application world and is a generator for a set, usually infinite, of instantaneous data bases for that application world. A data base model is a generator for an infinite set of structures every one of which can be regarded as an instantaneous data base for *a* state of *an* application world. (The model should be rich enough to provide a representation for every possible state of every reasonable application world.)

Data base models are supplied with general user languages. Some of them are query languages. A query is a specification of information which a user wants to extract or deduce from an instantaneous data base without knowing its exact extent. A query is interpreted as a partial function from instantaneous data bases to some data structures.

Among other data base languages are the update (transaction) languages. An update transaction expresses a transition between states of an application world, plus a query. It is interpreted as a partial function from instantaneous data bases to instantaneous data bases plus data structures containing information to be displayed. (Unlike the interpretation of queries, the interpretation of updates usually also depends on integrity and inference rules, fixed for the application, as discussed later.) These functions are not total when the implementing software may loop infinitely in some cases.

It is usually desired that data base user languages possess the following properties:

(1) They should be powerful enough to provide expressions for all "reasonable" requirements of users for any "reasonable" application world. A "reasonable query" must be physical-data-independent, at least in the following senses of [2]:

- Its output may not depend on the actual ordering of data in the physical data base. This is avoided by regarding a query as a transformation on an abstracted model of data bases, *e.g.* the relational data base model as defined in [3] (where an instantaneous data base is a collection of named n-ary *mathematical relations* over domains of values) or a semantic binary data model — [4], [5], [6] (where an instantaneous data base is a collection of named unary and binary relations).
- Its output may not depend on the physical representation of abstract objects in a data base. (In the semantic binary model, some objects are uninterpreted, representing real-world entities, and some are concrete values. There is no such clear distinction in the relational model since all the objects are logically represented there by values supplied by the user.) This principle may be extended by defining several types of objects:

  (i) the uninterpreted objects (the only meaningful mathematical operation on them is the binary function ''='' yielding a Boolean value),

(ii) fully-interpreted objects (e.g. integers, on which every partial recursive function may be meaningful), and

(iii) semi-interpreted objects, on which a collection of meaningful functions may be defined (e.g. the comparators >, <, etc. on names of people).

Any of these types may be empty for a particular application. The first two types are special cases of types of semi-interpreted objects. So, all the types can be collapsed into one by defining one set of meaningful functions from tuples of objects to objects or error elements.

(2) They should allow convenient expression of at least "frequent" requirements.

(3) They should be implementable by software.

There is no consensus on the extent of "reasonable" requirements of a query language's expressive power beyond the minimal data-independency. Unlike the language proposed herein, other proposals for query languages did not provide the possibility to express all meaningful queries and used a narrow interpretation of "reasonable requirements".

- These "reasonable requirements" are sometimes restricted to the queries expressible in Codd's Relational Algebra (or in the Relational Calculus), and thus languages whose expressiveness is equivalent to the Relational Calculus are called Codd-complete (*cf.* [1]).
- [7] showed that quite reasonable queries, such as those involving transitive closure are inexpressible in Codd's Algebra and proposed to extend the Algebra by a fixed-point operator.
- A more powerful class of languages, using Horn clauses, is advocated in [8] and [9]. A representative of this class is Prolog (*cf.* [10]). Incomplete expressibility of Horn clauses was shown in [11].
- [12] proposes a much richer language model, which supports all computable data-independent queries excluding those necessitating generation of new uninterpreted objects (e.g by an update transaction) and still keeping some restrictions on computations that necessitate interpretation of values. (Their language has a powerful capability of calculation of values, including aggregate calculations, e.g. counting, but does not allow all meaningful computations.) Another difference is that their language is highly-procedural.

  The exclusion of value-computations is argued by most authors by the desirability of enforcing the separation between data extraction (specified by a query) and data computation (specified by a program). Such separation may not always be justified, especially when one wishes to use queries for updates or for specification of inference rules.

An objective of this work is to design a query language which possesses the following properties:

(1) It is **absolutely complete**[1], i.e. every *computable* transformation is expressible in it. (A computable transformation is a partial recursive function of numbers effectively representing *sets* of tuples of objects.) (We use the term *absolutely* complete because of the the recent inflation in the meaning of "completeness" of query

languages, *e.g.* the languages equivalent to Codd's Algebra are often called "complete", while neither they, nor any other known non-procedural language but ours, could express all the reasonable queries.)

(2) It is user-friendly:

- The user states not how to extract the information, but what properties the extracted information should possess.
- No query needs to regard types of information which are irrelevant to it.
- Queries are independent of representation and of computer-oriented decisions.
- The users are enabled to exploit indeterminism.
- Both the semantic-oriented and the table-oriented user are provided with appropriate syntax.
- The user can easily specify calculations on values when needed. Arbitrary aggregate calculations (such as summation, counting, etc.) are also expressible.
- The language is provided with "syntactic sugar" to make it more "friendly" to the end-user.
- The same syntax can be used to specify update transactions and integrity and inference rules.

(3) The language is implementable. Of course, the implementation cannot be efficient due to the expressive power of the language. Heuristic techniques can be designed to implement efficiently some important subsets of the language. Whether or not the language is actually implemented by reasonably efficient software, the language can serve as:

- a formalism for reasoning about databases and for high-level specification of database operation before they are programmed in a lower-level language;
- a model for generation of efficiently-implementable *sub*languages; and
- a model for comparison of different languages.

(4) For any definition of data-independence expressed by a set of meaningful operations on objects, there is a restricted syntax of the language, which generates all and only the data-independent queries and transformations. This is clarified in the section entitled 'Data Independence'.

2. Basic Definitions

Query languages: computability and implementability

Definition. A query language is a tuple $\boldsymbol{L}=(Alph, IDB, \mathrm{L}, S_L)$, where:

- *Alph*, called the alphabet of the language, is a finite set;
- *IDB*, called the data base model, is a denumerable set; its members are called [instantaneous] data bases;
- L, called the syntax of the language, is a set of strings, i.e., a subset of $Alph^*$;

  every $q \in L$ is called a query.
- S_L, called the semantics of the language, is a partial function from (L x *IDB*) to *IDB*;

  When a query q is applied to an input instantaneous data base *idb*, the result is $S_L(q, idb)$.

  If $(q, idb) \notin domain(S_L)$ then S_L is undefined for (q, idb), i.e. the query q would loop for the input data base *idb*.

  The semantics of a query q is the partial function from *IDB* to *IDB* defined by $S_{L,q}(idb) = S_L(q, idb)$.

Auxiliary definition. A [partial] function f from a denumerable set U to a denumerable set V is called [partial] computable iff there exist a finite set of characters *Alph* and two-way-effective bijections $\beta_u : U \rightarrow Alph^*$ and $\beta_v : V \rightarrow Alph^*$ such that the function $\beta_v \circ f \circ \beta_u^{-1}$ is [partial] recursive.

Definition. A query language is implementable iff its semantics is a partial computable function.

Definition. A query language is [absolutely] complete if for every partial computable function f over its model there is a query whose semantics is f.

Abstraction of the data base model *IDB*

Let A be a fixed finite set of all printable characters.

Let Objects = Relationnames = D = A^*. We require that "true" and "false" be in D.

Some objects are numerals for numbers in decimal notation.

We use the following data base model:

$$IDB = \text{the set of all finite subsets of } Objects \times Relationnames \times Objects$$

Every triple in every instantaneous data base represents a binary relationship whose relation-name is in the middle. Binary relations are sufficient because the higher-ranked ones are meaningfully expressible by binary relationships, *e.g.* so:

$$\mathrm{P}[A_1 : a_1, A_2 : a_2, ..., A_n : a_n] \text{ iff } \exists z\; zA_1 a_1 \wedge zA_2 a_2 \wedge ... \wedge zA_n a_n \wedge z\,\mathrm{P}\, null.$$

Representation of n-ary relations by binary serves several purposes: it simplifies syntax of the queries in the language that we shall define and makes it more convenient to the user; it overcomes some update anomalies of relational data bases; it will simplify our formal notation and proofs, though it is not vital for their correctness.

3. The Proposed Language

The proposed language is a set of formulas, called queries, which are interpreted as partial transformations over the set *IDB* of instantaneous relational data bases. In the considered data base model *IDB*, introduced in Section 2, an instantaneous relational data base is a finite family of named finite relations over a fixed denumerable set *D*, called the domain of objects. (*D* can be further subdivided into domains of concrete mathematical values and domains of abstract objects.) When a instantaneous data base is transformed to another one (by a query or by an update), the former data base is called "the input" and the latter "the result".

The syntax and semantics of queries are formally defined in Appendix 3 (A and D respectively). Here we describe them in an informal way because that may suffice to many readers.

The **semantics of a query** is defined in two steps: **First**, the formula is assigned with an **assertional interpretation**, which is a partial predicate over *IDB*. **Then**, a **transformation** is derived from it. It transforms any input data base into a *result* such that there exists a data base, called a **virtual data base**, which:

- consists of three distinguishable parts: the input, the *result*, and temporary data. We mean by this that there are two fixed unary functions, "input" and "result" from *IDB* to *IDB*. Then the input part of *idb* is *input*(*idb*), the result part of *idb* is *result*(*idb*), and the temporary data is what remains when we subtract the input and result parts from *idb*;
- **satisfies the assertion;**
- is **minimal for this input:** every other data base included in it and having the same input part contradicts the assertion;
- is (**non-deterministically**) chosen if other "minimal" data bases exist.

The result can be undefined if all the (virtual) data bases, in which the input is embedded, contradict the assertion, or if for every virtual data base satisfying the assertion there is a sub-data-base (i.e. a subset thereof) for which the predicate is undefined.

The following describes the abstracted syntax of queries (before user-friendly "sugar"), yielding their assertional semantics. A query is expressed as a closed formula in an applied first order predicate calculus. For any (virtual) database the formula is interpreted as *true, false* or *undefined*. The formula is composed of:

- constants, which are any objects of *D*, not necessarily in the virtual data base;
- quantified variables ranging over the set of objects which appear in the virtual data base;
- a unary predicate symbol interpreted as the equality of its argument to the object 'true';
- a predicate symbol interpreted as the belonging of a tuple of objects to a named relation in the virtual data base; (The objects are evaluated from terms. The relation-

name is usually specified as a constant, but some rather "unreasonable" queries [see Theorem 4] may necessitate an evaluation of the name from a term.)

- operators: "and", etc; (In the principal variant of the language, three-valued parallel logic is used.)
- function symbols expressing scalar mathematical operations over the domain of objects, including comparators (>,<, etc.) yielding Boolean values.

Two alternative variants of the language have been developed. The first one, focuses the assertional semantics on data base structure, while the scalar mathematical operations are expressed, for the sake of separation from information-manipulation, by an infinite set of function symbols syntaxed as recursive functional expressions having the least-fixed-point semantics. This extension of the set of functional symbols does not contribute to the expressive power of the language because every scalar function can be expressed by a logical assertion using only a fixed finite set of standard functions. In the other proposed variant [13], only a finite set of "cataloged" function symbols is used, while the rest of computable scalar operations are generated from them exploiting the principal transformational-assertional semantics of the language. A very large class of transformations can be specified *without* function symbols at all except the equality symbol "="; this includes all those queries definable by: Codd's Algebra (without use of comparisons of objects; otherwise they are specifiable using one functional symbol ">"); Codd's Algebra extended by the fixed-point operator; Horn clauses; queries inexpressible by Horn clauses, e.g. Example 3 of the next section; and many other classes.

Sublanguages have been investigated where some function symbols are used, while others are prohibited in order to maintain data-independence. In a special case the domain of objects is split into concrete objects and abstract objects. Only "=" is defined on abstract objects, and a full function space is defined on the sub-domain of concrete objects.

The language is based on the abstracted syntax specified above and on syntactic "sugar" — user friendly abbreviations of formal expressions. A complete "sugar" is specified in [13]. Among these "sugar" abbreviations are: a full scope of logic operators; contextual defaults for quantifiers; abbreviations of sub-assertions expressing aggregate application of associative scalar functions, *e.g.*, summation, counting etc; substitution of variables by *examples* of objects (as in *Query-By-Example*); representation of relationships by simple English phrases; distinct syntax variants for the Relational data base model and the Semantic Binary data base model. In the following section, some examples of queries with this "sugar" are given.

4. Examples of Queries

The examples use the following semantic binary schema, specifying categories (unary relations) as squares and binary relations as arrows.

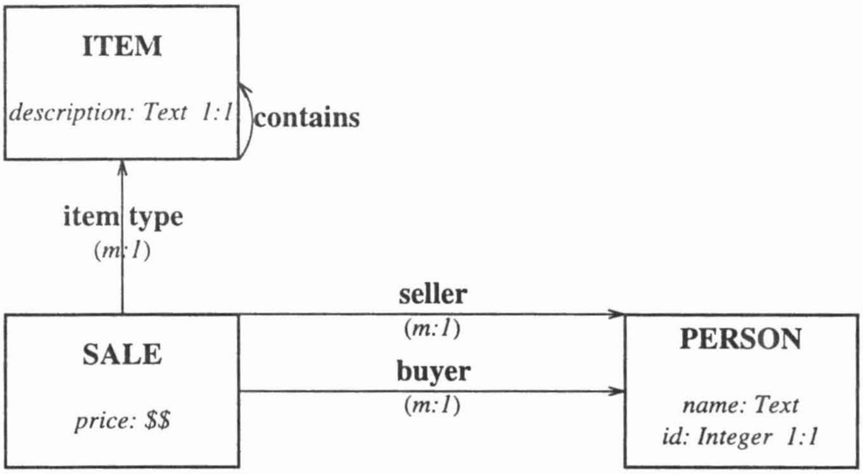

Figure 4-1. The semantic database schema used in the examples. A *sale* is a transaction of a merchandise of the *item-type* for the *price* between the *seller* and the *buyer*. The many-to-many relation *contains* forms a bill of materiel of item-types.

Example:

An example of using a transitive closure.

/* who bought a bolt directly or indirectly, *e.g.*, bought a lock, door, train car, train, etc.? */

∀b,s,c: (if (b BUYER-*input* s) ∧ (b ITEM-TYPE-*input* c) then (s GOT c)) ∧

∀s,c,d: (if (s GOT c) ∧ ((c COMPONENT-*input* d)) then (s GOT d)) ∧

∀s,x,n: (if ((x DESCRIPTION-*input* 'bolt') ∧ (s NAME-*input* n)) ∧ (s GOT x) then ((n GOT-A-BOLT*result*)).

Example continued.

The same query using the standard sugar of the language:
if given:
somebody *is the* BUYER *of a* bargain,
car (*e.g.*,) is the ITEM-TYPE *of the* bargain
then somebody GOT a car *and*
if somebody GOT a car (*e.g.*,) *and*

given: door (*e.g.*,) *is a* COMPONENT *of* car
then somebody GOT a door *and*
if somebody GOT something *and*
given: the DESCRIPTION *of* something *is* 'bolt',
the NAME *of* somebody *is* smith (*e.g.*,)
then result: smith 'GOT A BOLT'

Example:

Table-oriented specification.

Table-oriented users would prefer a relational schema as follows:

relation PERSON [ID, NAME];

relation SALE [BUYER-ID, SELLER-ID, PRICE, ITEM_DESCR];

relation ITEM [DESCRIPTION];

relation COMPONENT [CONTAINING_DESCR,
CONTAINED_DESCR]

The above query could be formulated by them as follows:

```
if given:
    SALE [BUYER-ID: buyer, SELLER-ID: seller,
        PRICE: price, ITEM-DESCR: item]
    then GOT [OWNER: buyer, THING: thing] and
if GOT [OWNER: owner, THING: thing] and
    given: COMPONENT [CONTAINING_DESCR: thing,
                CONTAINED_DESCR: otherthing]
    then GOT [OWNER: owner, THING: otherthing] and
if GOT [OWNER: owner_id, THING: bolt] and
    given: PERSON [ID: owner_id, NAME: name]
    then result THOSE_WHO_GOT_A_BOLT [NAME: name]
```

Example:

A query which cannot be specified by Horn clauses.

/* What items have no less components than the item described as 'car'? */

(This query does not use function symbols.)

given : 'car' *is the* DESCRIPTION *of* c (*e.g.*), watch (*e.g.*) *is the* DESCRIPTION *of* w *and*

result : watch HAS-MANY-COMPONENTS *and*

if stone (*e.g.*) *is* PAIRED-TO wheel (*e.g.*) *then given* : stone *is a* COMPONENT *of* watch, wheel *is a* COMPONENT *of* car *and*

if stone *is* PAIRED-TO wheel *and* stone *is* PAIRED-TO x *then* x=wheel
and

if wheel *is a* COMPONENT *of* car *then exists* stone *s.t.* stone *is*
PAIRED-TO wheel.

Example:

/* find every seller's total income */
if given: man *is a* PERSON, *the* NAME *of the*
man *is* smith
then exits income *s.t.*
(income *is the sum of* PRICE *of*
bargain *dependent on* man *s.t.* (man *is the* SELLER
of the bargain)) and
result: *the* INCOME *of* smith *is* income.

Note: the above phrase "the sum of", which might look aggregate and second order, is a syntactic sugar abbreviation for a longer *first-order non-aggregate* phrase using only one binary function symbol "+", which is applied to pairs of integers denoting prices [13].*/

Example:

/* Update: Boards are not components of processors, but vice versa. */
if *given* :
'Processor' *is the* DESCRIPTION *of* p,
'Board' *is the* DESCRIPTION *of* b
then

result delete : b *is a* COMPONENT *of* p *and*

result insert : p *is a* COMPONENT *of* b

Example:

Example of concrete minimal syntax *without "sugar"*. (This is actually the intermediate syntax obtained after translation of a user-oriented language.)

/* Find the prices of all the items */

∀price ∀sale ∀item ∀itemdescription
(IsaRelationship (sale 'PRICE-input' price) ⊃
(IsaRelationship (sale 'ITEM-TYPE-input' item) ⊃
(IsaRelationship (item 'DESCRIPTION-input' item-description) ⊃

(IsaRelationship (item-description 'ITEM-PRICE-result' price))))

5. Data Independence

Several **submodels** of the language are derived in [13]. They are intended to restrict the use of undesirable or meaningless operations on objects. One of the most important cases differentiates between:

- abstract objects, representing real-world entities in the Semantic Binary Database Model; (No mathematical operation such as "+" or "<" is meaningful on them.)
- concrete values, which are mathematical objects representing themselves, e.g. '234.35', 'abcd'. (They have the same meaning in the real world and in the computer.)

A more general case is parametrized by a family of permitted operations on the domain of objects and its subdomains. The submodels are proven to be able to express every data base transformation reasonable within the restrictions parameterizing the submodels.

A finite set of basic function symbols is sufficient to have the complete power of the language. The rest of partial recursive functions ($[D^* \rightarrow D]$) can be expressed using the postconditional semantics of the query language. Unlike that ''saving'' in function symbols while keeping the complete power, in the following we wish to actually restrict the power of the language by removing from it the ability to specify computations which are meaningless and should be forbidden in a user's system of concepts.

The general case to be investigated is the one in which a user is provided with a family of functions on values considered meaningful for a given data base or a data base management system. This family does not necessarily contain a basic set sufficient to create all the computable functions over the domain of objects using the programming power. The functions may be partial.

Families of functions of special interest are those differentiating between *abstract objects* and *concrete values*. On the subdomain of the abstract objects there are only two meaningful functions: the characteristic function *is-abstract* giving *true* for abstract objects and *false* for concrete values, and the binary function *equality* giving *true* or *false* for pairs of objects. The rest of such a family is a basic set of functions on the subdomain of concrete values. Using this basic set and a program control power, every computable function on the subdomain of values can be expressed.

In the following, let Φ be a family of operations on the infinite set of all possible objects D. Every member of Φ is a function from D^* to $D \cup \{undefined\}$. (Φ is not necessarily a special case like the one described in the previous paragraph.) D is assumed to contain the special objects $\{error, true, false\}$.

$$D^* = D^0 \cup D \cup D^2 \cup D^3 \cdots$$

Though binary operations are sufficient to have the complete power of the language, we are aiming to restrict the power and to be able to model exactly any practical restriction.

That's why we permit here n-ary operations — some of them cannot be generated from binary ones without choosing them strong enough to permit generation of functions which are beyond a desired restriction.

Let L_{Φ} be the language as defined above but using only function symbols from Φ (and no recursion.)

I claim, intuitively, that L_{Φ} has all the power reasonable within the restriction of Φ, including:

(a) the ability to generate every function computable using program control and the set of operations Φ;

(b) the ability to generate vertical functions, such as *sum* or *average* of values, i.e. to relate some objects to applications of functions (a) on sets of values;

(c) the ability to create new objects, including abstract objects;

(d) the ability to specify every data base transformation, which does not necessitate interpretation of objects beyond what can be done using the functions Φ.

These and other objectives will be specified rigorously after I define isomorphism of data base transformations.

In addition to Φ, queries of L_{Φ} may use constant symbols. But I claim (so far intuitively) that a query needs to use only those constants which are absolutely relevant to its purpose, i.e. any program would have to use these constants in addition to Φ.

The use of constants is not redundant, i.e. the constants cannot be substituted by 0-ary functions from Φ, because:

1) The set Φ is fixed for the language L_{Φ} due to global restrictions which in a given data base or DBMS are desired to be imposed on all queries.

2) Not all permitted constants can be generated from Φ when it is intentionally more restricted.*E.g.*, when social security numbers are considered, only their comparison is permitted in Φ, but we would certainly wish to permit asking a query inquiring about any specific social security number, appearing as a constant in the query. Usually, the permitted constants are all nonabstract objects.

3) If instead of Φ we were fixing (globally for the language) a richer set containing (or able to generate) all the permitted constants, which is generally an infinite set, then every query would become undesirably less free and more deterministic due to fixed interpretation of constants which it does not need.

Now I shall formalize the discussion.

Definition

A bijection $\iota : D \rightarrow D$ is called a Φ - isomorphism iff

$$\forall (d_1, \ldots, d_n) \in D^* \quad \forall f \in \Phi \;\; f(\iota(d_1), \ldots, \iota(d_n)) \equiv \iota(f(d_1, \ldots, d_n))$$

(Note: the $\equiv$ symbol covers the case when both sides are undefined.)

Definition

For a given instantaneous data base *db*, a Φ-isomorphism ι is called *db-preserving* iff for every object *d* appearing in *db*,

$$\iota(d) \equiv d$$

Definition

A computable data base transformation $\phi: IDB \xrightarrow{p} IDB$ is Φ-preserving if for every $db \in IDB$ and for every Φ-isomorphism ι, $\phi(\iota(db)) \equiv \iota(\phi(db))$.

Definition

Two data base transformations ϕ,ψ are called Φ-isomorphic iff for every instantaneous data base $db \in IDB$ there exists a *db*-preserving Φ-isomorphism ι such that $\phi(db) \equiv \iota(\psi(db))$.

Definition

Let C be a finite set of constants, a subset of D. Two data base transformations are called (Φ, C)-isomorphic iff they are $(\Phi \cup C')$-isomorphic, where C' is the set of constant functions equivalent to C. A Φ-isomorphism ι is called (Φ,C)-isomorphism if it is a $(\Phi \cup C')$-isomorphism.

Proposition:

For every finite set of constants $C \subset D$, every computable (Φ,C)-preserving data base transformation is expressible up to a (Φ,C)-isomorphism in L_Φ with C, *i.e.*, for every such transformation ϕ there exists a query $q \in L_\Phi$ using no other constants but C, whose semantics ψ is (Φ,C)-isomorphic to ϕ.

Corollary

Every computable Φ-preserving data base transformation is expressible in L_Φ up to an isomorphism,

> *i.e.*, for every such transformation there exists a query $q \in L$ whose semantics ψ is Φ-isomorphic to ϕ, and the query q uses no constant symbols.

6. Summary of Main Theorems About the Language

The following is a review of main results about the proposed language model. The proofs of the most important results are included in Appendices of this paper. All the listed results are proven in detail in [13].

1) The language is *implementable*, i.e. it has an interpreter.

2) The language is *absolutely complete*, i.e. for every partial computable function $\phi: IDB \rightarrow IDB$ there exists a query $q \in L$ whose semantics is ϕ.

3) The sublanguage containing only *deterministic* queries is absolutely complete too. Thus, the non-determinism (being desirable for user-friendliness and implementation optimization) is not the reason for absolute completeness.

4) Every query whose result can be affected only by a finite set of relation-names, i.e. whose intrinsic meaning does not necessitate quantification over the set of names of relations (as can be for Data Base Administrator's queries) can be specified using only

constants as names of relations. (*I.e.* the language can be seen syntactically as *first-order* with relations as predicate symbols.)

5) A standard *finite set of function symbols* defined on the domain of objects is sufficient for absolute completeness of the language. The other functions on values can be represented by assertions, although such representations can be undesirable from a methodological point of view.

6) If the language is further restricted to any set of standard function symbols on values (in order to permit only meaningful operations on some domains, *e.g.* only equality-verification on abstract objects), then every query meaningful within this restriction is expressible in the restricted language up to an isomorphism.

7) The language can be used to specify every *update* transaction.

8) The language can be used to specify every *integrity* and *inference* rule in the data base.

9) There is a semantic extension of the language (without alteration of the syntax) to cover the behavior of queries and update transactions in the presence of integrity and inference rules.

Acknowledgment

This research was supported in part by NASA (under grant NAGW-4080), ARO (under BMDO grant DAAH04-0024), NATO (under grant HTECH.LG-931449), NSF (under grant CDA-9313624 for CATE Lab), and State of Florida.

APPENDICES

Appendix 1 - The Implementability Theorem

The proof of implementability is *sketched* here by defining an implementation of a very high complexity. In practice a heuristic implementation is needed for the language or its sublanguages.

1. There is a procedure to implement the predicate

   VERIFY (q, vdb, idb)

   ("does the virtual data base vdb satisfy the assertion q?").

The procedure acts as follows. First it checks whether idb is the given part of vdb. If not, it halts with false. Otherwise it continues. Quantifiers are resolved yielding a finite number n of atomic formulae connected by logical operators (this is because vdb is finite and all the quantifiers range over its objects). n parallel processes are issued to evaluate the clauses. These processes are correlated so that a halting process will cause an abortion of those processes whose results will not influence the interpretation of the assertion (as defined by the three-valued logic below).

2. An effective inclusion-preserving enumeration E of the set IDB of all instantaneous data bases is constructed.

3. The following is a procedure to evaluate a query. The inputs are: $q \in L$, $idb \in IDB$.

The procedure uses an unlimited quantity of parallel processes, but at every instant of time the number of processes is finite, and thus they can be implemented by one sequential process.

Let $vdb_1, vdb_2, \ldots, vdb_n, \ldots$ be the inclusion-preserving enumeration of IDB constructed in (2).

Let Q be a fixed quantity of time.

Let BUFFER be an unlimited, initially empty, interprocess storage (which will contain indices of virtual data bases found as contradicting assertion q).

At the beginning, the first process PR_1 is invoked.

Every process PR_n acts as follows after its invocation:

```
A    — Start computing VERIFY ($q, vdb_n, idb$) until "local" time Q elapsed.
B    — Invoke the process $PR_{n+1}$.
C    — Continue computing VERIFY ($q, vdb_n, idb$) until true of false is obtained or
       forever (unless externally aborted).
D    — If false has been obtained then:
     D1   — Insert the index $n$ into BUFFER;
     D2   — Loop forever (unless externally aborted).
E    — If true has been obtained then:
     E1   — If every proper subset of $vdb_n$ is in BUFFER, then:
          E2   — Output the "result part" of $vdb_n$;
          E3   — Abort all the processes, including the current process.
          else: repeat $E_1$ (forever or until internally or externally aborted).
```

Appendix 2: Completeness Theorems.

A. Absolute completeness of the maximal language.

Theorem. The maximal language L defined above (where Φ contains every partial recursive function from D^* to D represented by a recursive expression) is absolutely complete, i.e., for every partial computable function $\phi: IDB \to IDB$ there exists a query $q \in L$ whose semantics is ϕ.

The proof is preceded by its sketch.

A query q is constructed whose semantics is ϕ. The assertion of the query consists of three subassertions:

- an assertion implying existence of a special object in the virtual data base encoding the whole input data base,

- an assertion implying existence of an object encoding the resulting data base,
- and an assertion relating these two objects by a derivative of ϕ.

These assertions are constructed so that the following is insured:

- the query will be deterministic (to be used in the next theorem);
- the query is convertible into an appropriate query for the language LL not using variables or expressions as names of relations (to be used in "LL almost completeness" theorem);
- the conjunction of the assertions is undefined if and only if ϕ is undefined for the input data base, provided the evaluation is done by parallel communicating processes;
- the conjunction gives false for every subset of the desired virtual data base.

Proof:

Let ϕ: $IDB \rightarrow IDB$ be a partial computable function.

1) Encode IDB by D.

Let $*: D \times D \rightarrow D$, sc: THE-SET-OF-ALL-FINITE-SUBSETS-OF(D) $\rightarrow D$ be two two-way effective bijections (existence of which is well known). Let tr: $IDB \rightarrow D$ be the two-way effective bijection defined by:

$tr(db) = sc(\{r*(\alpha*\beta) \mid (\alpha\ r\ \beta) \in db\})$.

Let $f = tr \circ \phi \circ tr^{-1}$. By the Theory of Computability, f is a partial recursive function from D to D.

2) Define total recursive functions from D^2 to D simulating set operations:

$insert(s,d) = sc(sc^{-1}(s) \cup \{d\})$

$remove(s,d) = sc(sc^{-1}(s) - \{d\})$

$in(d,s) =$ if $d \in sc^{-1}(s)$ then '*true*' else '*false*'

3) Abbreviate:

∅-code — the constant representing $sc(\varnothing)$ (i.e. the constant encoding the empty set.)

f, ***in***, ***insert***, ***remove*** — recursive expressions representing the corresponding functions f, in, $insert$, and $remove$.

GIVEN (x, r, y) — IsaRelationship(x_1, r, y), where x_1 is an expression concatenating the string 'input' to the value of x (i.e. GIVEN is a predicate stating that a tuple belongs to the input part of the virtual data base).

RESULT (x, r, y) — analogously.

TEMP (x, r, y) — IsaRelationship(x, y, z) (to be used for tuples which are neither in the input part nor in the result part of the virtual data base.)

4) The query q.

The following sentence abbreviates the assertional syntax of the query and is composed of clauses (marked C_i), each of which is preceded by a comment (enclosed in /*...*/) outlining the subassertion expressed by the clause. The names of the unary relations (categories) of the virtual data base are given in enlarged italics.

/* C_0 and C_1: there is a temporary object encoding the whole input data base */

/* C_0: there is a temporary object encoding the empty set */

```
TEMP (∅-code ENCODES-A-SUBSET-OF-THE-INPUT-DB) and
```

/* C_1: for every existing code of a subset and for every triple in the input db, there is a temporary object encoding that subset enriched with this triple */.

```
∀setcode , x ,y ,r
    if TEMP (setcode ENCODES-A-SUBSET-OF-THE-INPUT-DB)
       and GIVEN (x ,r ,y ) then
       TEMP (insert (setcode ,(r * (s * y )))
         ENCODES-A-SUBSET-OF-THE-INPUT-DB) and
```

/* C_2: there is a temporary object which equals f (the encoding of the whole input data base); this object should encode the whole result */

```
∀inputdbcode
    if (∀x ,r ,y if GIVEN (x ,r ,y ) then
              IsTrue(in ((r * (x * y )), inputdbcode )))
      then TEMP (f (inputdbcode )
              ENCODES-A-SUBSET-OF-THE-RESULT) and
```

/* C_3: the result is actually what is encoded by the above object */

```
∀setcode
    if TEMP (setcode ENCODES-A-SUBSET-OF-THE-RESULT) then
```

/* $C_{3.1}$: the encoded set is either empty or contains a resulting triple */

```
((IsTrue (setcode = ∅-code ) or
  ∃x ,r ,y (RESULT (x r y ) and
      IsTrue(in ((r * (x * y )), setcode )))) and
```

/* $C_{3.2}$: inductively, every triple contained in the set must be in the result; but using the above we invert this thus: */

```
∀x ,r ,y
  if RESULT (x r y ) then
    TEMP (remove (setcode , r * (x * y ))
        ENCODES-A-SUBSET-OF-THE-RESULT))
```

5) Let $\bar{q}$ be the semantics of q. The following proves that $\bar{q} = \phi$.

Let $idb \in IDB$. Consider two cases:

(i) $\phi(idb)$ is undefined.
It has to be shown that $\bar{q}(idb)$ is also undefined. Assume the contrary. Then there exists $vdb \in IDB$ satisfying the assertion and containing idb. By definition of the "parallel and", all the four clauses are interpreted to *true* for vdb. $C_0 \wedge C_1$ imply inductively that there exists $inputdbcode = tr(idb)$ in vdb. This and C_2 imply that there is $f(inputdbcode)$ in vdb. Thus $f(tr(idb))$ is defined and so is $\phi(idb)$, in contradiction to the assumption. Thus $\bar{q}(idb)$ is undefined.

(ii) $\phi(idb)$ is well-defined (not *undefined*).
Let vdb be as follows: its input and result parts are idb and $\phi(idb)$ respectively, and its remainder consists of two instantaneous unary relations: ENCODES-A-SUBSET-OF-THE-INPUT-DB is $\{tr(S) | S \subseteq idb\}$, ENCODES-A-SUBSET-OF-THE-RESULT is $\{tr(S) | (S \subseteq \phi(idb)\}$.

vdb satisfies the assertion. It remains to show that every one of its proper subsets containing idb contradicts the assertion.

Assume the contrary. Let $idb \subseteq vdb' \subset vdb$ so that vdb' does not contradict the assertion. Then the interpretation of the assertion for vdb' is *true* or *undefined*.

Consider both cases:

(a) The interpretation is *true*. Then idb is the "input part" of vdb' and all the four clauses yield *true* for vdb'. $C_0 \wedge C_1$ imply that the instance of ENCODES-A-SUBSET-OF-THE-INPUT-DB in vdb' includes $\{tr(S) \mid S \subseteq idb\}$. Thus, $tr(idb)$ is contained in this instance.

Then, by C_2, $f(tr(idb))$ is in the instance of ENCODES-A-SUBSET-OF-THE-RESULT. Then, by C_3, the result part includes $\phi(idb)$ and the instance of ENCODES-A-SUBSET-OF-THE-RESULT includes $\{tr(S) | S \subseteq \phi(idb)\}$. Thus, $vdb \subseteq vdb'$, in contradiction.

(b) The interpretation is *undefined*. Then, by definition of "parallel and" at least one clause yields *undefined* for vdb' and no clause yields *false*. All the clauses, except C', involve only total functions. Thus, C_0, C_1 and C_3 yield *true* and C_2 yields *undefined*.

The "input part" of vdb' is idb (otherwise the assertion would yield *false*). This and $C_0 \wedge C_1$ imply that there is $tr(idb)$ in the instance of ENCODES-A-SUBSET-OF-THE-INPUT-DB. But $f(tr(idb)) = tr^{-1}(\phi(idb))$ is defined. (Possibly there is another setcode in the above relation's instance such that $f(setcode)$ is *undefined* and setcode encodes a set containing all the triples of "the input part".) After the resolution of quantification, C_2 is a conjunction of many clauses, none of which yields *false* (otherwise C_2 would yield *false*). Thus, since $f(tr(idb))$ is defined, the subclause for $tr(idb)$ must yield *true*. Thus, $f(tr(idb))$ is in the instance of ENCODES-A-SUBSET-OF-THE-

RESULT. Continuing the reasoning analogous to that of **(a)**, we get: $vdb \subseteq vdb'$, in analogous contradiction.

B. The completeness of deterministic queries.

Theorem. The sublanguage of L containing only deterministic queries is also absolutely complete.

Proof: Following the proof of the previous theorem, we find that if there is vdb' satisfying the assertion, then $vdb \subseteq vdb'$. Thus, no other but vdb can be chosen; so the query is deterministic.

C. Almost-completeness of The First-Order Sublanguage

I shall prove here that any query can be stated so that relations are named only by constants, unless the query must deal with infinitely many relevant relation-names (which usually would be meaningless in an end-user's query).

Definition. A set $S \subseteq D$ contains all relation names *relevant* for ϕ: $IDB \overset{P}{\rightarrow} IDB$ iff:

- $\{r \mid \exists idb \in dom(\phi), \exists x, y \in D : (x\ r\ y) \in \phi(idb)\} \subseteq S$, i.e., S contains every relation-name appearing in some output, and
- for every $idb \in IDB$

  $$\phi(idb) \equiv \phi(idb - D \times (D - S) \times D)$$

  ("$\equiv$" means that either both sides are undefined or they are equal).

Definition. A function ϕ: $IDB \overset{P}{\rightarrow} IDB$ has a *finite* set of relevant relation-names iff there is a finite set which contains all the relation-names relevant for ϕ.

Note:

1) Transformations which do not have such a finite set intuitively do not represent specific needs at the application level but rather something at the DBMS level. For example, copy the whole data base, estimate its extent, list its relation-names.

2) The semantics of update transactions was defined using "delete" and "insert" with respect to a pretransaction state. Thus, the above intuitive claim is also true for transactions.

Theorem. The language LL (i.e., those queries of L which use only constants as names of relations) generates all the partial computable functions from IDB to IDB having finite sets of relevant relation-names.

Proof: Let ϕ be a partial computable function from IDB to IDB having a finite set S of relevant relation-names. Denote the elements of S by $r_1, r_2, \ldots, r_n$. From the structure of the query q defined in the proof of the principal completeness theorem, obtain a query q_2 by resolving all the quantifications of the variable "r". Thus, "$\forall r\ \tau$" is transformed to "$\tau_1 \wedge \tau_2 \wedge \ldots \wedge \tau_n$" where τ_i is τ in which r is substituted for the constant representing r_i. (Respectively, "$\exists r\ \tau$" is transformed to "$\tau_1 \vee \ldots \vee \tau_n$".)

Let $\bar{q}_2$ be the semantics of q_2. We will show that $\bar{q}\,' = \phi$.

Let $idb \in IDB$.

Consider the following cases:

1) All the relation-names appearing in *idb* belong to S. So do the relation-names of $\phi(idb)$, provided this exists. The assertions q and q_2 are interpreted equivalently, and thus the queries must yield the same results (or *undefined*).

2) There is a relation-name r_0 appearing in *idb* and not belonging to S. Following the proof of the principal completeness theorem, we find that $\bar{q}\,'(idb) \equiv \phi(idb - D \times (D-S) \times S)$ which in turn, by the condition of the theorem and the definitions above, is equivalent to $\phi(idb)$.

Thus, in every case $\phi(idb) \equiv \bar{q}\,'(idb)$.

Appendix 3: Syntax and semantics of the Language

This section is composed of the following:

- abstracted syntax of the language
- concretization of symbols
- abbreviations used to improve the readability
- semantics of the language

Intuitively, every query is an assertion about a virtual data base in which there are three distinguished parts: an input instantaneous data base, a desired output instantaneous data base (for the input) and temporary data. (The parts are distinguished by suffices of names of relations.)

A. Abstracted syntax of queries (and abstracted semantics of symbols)

A query is a closed formula in an applied first-order predicate calculus using the following disjoint decidable sets of symbols belonging to A*:

1. A set of constants effectively representing the set of all objects D, i.e., there is an effective bijection between the sets.
2. A denumerable set of variables.
3. A set Φ of functional symbols.

In the first variant of the language, we let the set Φ be an infinite set effectively representing the set of all partial recursive functions from $D \cup D^2$ to D. (Effectiveness means here that there exists a procedure which for given function symbol and argument objects gives the application of the corresponding function.)

Every term is a constant, a variable, or is the application of a function symbol to one or two terms.

Note: the Boolean functions are covered because:

"true," "false" $\in$ D.

4. Two predicate symbols: IS-TRUE (unary) and Is-a-Relationship (ternary). Atomic formulae are composed of application of IS-TRUE to a term or of application of Is-a-Relationship to a triple of terms.

We also define a restricted language LL in which the second element (representing a relation-name) in the latter triple must be a constant. (It is shown that LL is almost as powerful as L: all the queries for which there is a finite number of relevant relation-names can be represented in LL.)

5. The implication symbol "⊃", the universal quantifier "∀", the parentheses, and the blank. Atomic formulae are connected to form formulae using these symbols.

B. Concretization of symbols

6. Constants are strings over A obtained from objects of D thus: enclose the string in quotes (') (if the string already contains a ('), replace with (''), e.g. I'm becomes 'I''m'.)

7. Variables are strings containing no blanks and starting with a lower-case letter.

8. In the first variant of the language, the function symbols are defined by recursive functional expressions as follows. There is a finite set BASIC-FUNCS, called the set of basic function symbols, and containing at least the following elements: the strings APPEND, EQUAL, IF-TRUE-THEN-NULL-ELSE.

   The rest of the function symbols have the following form:

$$\textbf{(FUNCTION } \phi(\alpha,\beta) = \tau)$$

   where:

ϕ ,α, β are strings containing no blanks which are not constants, variables or basic symbols;

τ is obtained from a term by substitution of some occurrences of a variable for α, of another variable for β and maybe of some occurrences of a basic function symbol for ϕ;

β may be omitted.

In the first variant of the language, I have chosen to represent the scalar functions by recursive expressions because their procedural semantics is well-known and because they are quite user-friendly, and in order to clearly distinguish between the logical manipulation of the database structure in predicate calculus and arithmetics on scalars. However, the choice of the representation by recursive expressions is not crucial to the language. (It is shown in [13] that the infinite Φ is not needed for the completeness of the language: it is sufficient that Φ=BASIC-FUNCS. However, from the methodological point of view and from the point of view of user friendliness, the infinite Φ is preferable.)

Example:

Example of concrete unsugared syntax

```
 /*    find the ages of John's sons.  (Comment.)*/
       ∀man ∀son ∀age ∀name
               (Is-a-Relationship (man 'NAME given' 'John') ⊃
               (Is-a-Relationship (son 'FATHER given' man) ⊃
               (Is-a-Relationship (son 'AGE given' age) ⊃
               (Is-a-Relationship (son 'NAME given' name) ⊃
               (Is-a-Relationship (age 'IS THE AGE OF JOHN''S SON - result'
                         name) ) ) ) )
```

C. Readability "sugar"

To improve the readability of queries we define the following abbreviations:

9. The existential quantifiers $\exists$ and $\exists\,!$ are standardly derived from $\forall$. The universal quantifiers whose scope is the entire query may be omitted.

10. Abbreviations for operators between formulae:
    "$\sim\alpha$" stands for "$\alpha \supset$ IS-TRUE ('false')" "not" stands for "$\sim$"
    "if α then β" stands for "$\alpha \supset \beta$"
    "$\alpha \vee \beta$", "α *or* β" stand for "$\sim\alpha \supset \beta$"
    "$\alpha \wedge \beta$", "α and β" stand for "$\sim(\sim\alpha \vee \sim\beta)$"
    "if α then β else γ" stands for "$(\alpha \wedge \beta) \vee (\sim\alpha \wedge \gamma)$"
    "α iff β" stands for "$(\alpha \supset \beta) \wedge (\beta \supset \alpha)$"

11. "IS-TRUE (α)" may be abbreviated as "(α)"

12. "Is-a-Relationship ($\alpha\ \beta\ \gamma$)" may be abbreviated by "TEMP ($\alpha\ \beta\ \gamma$)" or just "($\alpha\ \beta\ \gamma$)". γ may be omitted if it is the constant standing for the null-string (to signify that the relationship is actually unary.) If β is a constant representing a relation-name having no other characters but capitals and the underscore, it may be written without quotes.

13. Any virtual data base is considered as consisting of three parts: given, result, and temporary. Intending to this we abbreviate:

    "GIVEN ($\alpha\ \beta\ \gamma$)" stands for "TEMP (α APPEND(β, 'given') γ)",

    "RESULT ($\alpha\ \beta\ \gamma$)" stands for "TEMP (α APPEND (β, 'result') γ)".

14. When no ambiguity arises, an argument triple ($\alpha\ \beta\ \gamma$) for TEMP may be written as: "(γ is a/the β of α") or "(the/a β of α is γ)." If γ is null, *i.e.* a unary relationship is represented by α and β, then ($\alpha\ \beta$) may be written (α is a β).

15. "(given: $w_1, w_2, \ldots, w_n$)" stands for:
    GIVEN (w_1), $\wedge$ GIVEN (w_2) $\cdots \wedge$ GIVEN (w_n),

    and analogously for "result."

Example:

(given: john is a BOY, the AGE of john is twelve)
stands for:
GIVEN (john 'BOY' '') $\wedge$ GIVEN (john 'AGE' twelve)

16. For the convenience of table-oriented users (the Relational Model),

$$"(\exists z\ (z\ \beta_1\ \gamma_1) \wedge (z\, \beta_2\, \gamma_{2)} \wedge \cdots\ (z\ \beta_n\ \gamma_n) \wedge (z\ \beta_0)\ )"$$

may be written as:

$$"\beta_0\ [\beta_1 : \gamma_1\ , \beta_2 : \gamma_2\ , \ldots ,\ \beta_n : \gamma_n\ ]",$$

when no ambiguity arises.

Example:

SALE [SELLER: john, BUYER:mike, ITEM:book]
stands for:
$\exists$ deal (deal SALE) $\wedge$ (deal SELLER john) $\wedge$
(deal BUYER mike) $\wedge$ (deal ITEM book)

Note: Inspired by Zloof's Query-By-Example, we use variables intuitively as examples of objects.

17. Function symbols may be written in prefix, infix, or postfix form. Also there is the following abbreviation: "IF α THEN β ELSE γ" stands for: "APPEND (IF-THEN-NULL-ELSE (α, γ), IF-THEN-NULL-ELSE (NOT(α), β))"

18. Constants representing objects containing only digits may be written without quotes.

Example:

Example of syntax with "sugar."

Note: In this example the variables are named to exemplify their contents.

/* For every adult whose mother is older than his father, find the factorial of the difference in ages of the parents. (Relations FATHER, MOTHER, AGE, and SURNAME are used.)*/

if (twenty $\geq$ 18) *and* (forty-three > forty) *and*
given:
AGE of john is twenty,
AGE of sarah is forty-three,
AGE of mike is forty,
mike is the FATHER of john,

```
                sarah is the MOTHER of john,
                SURNAME of john is smith
    then
        result: six is the
                 'factorial of the difference of the ages of the parents'
                 of smith
        and
           six =
           (FUNCTION FACTORIAL(x) =
              IF (x = 1)
                 THEN 1
                 ELSE x × FACTORIAL (x–1) )
           (forty-three  –  forty)
```

D. FORMAL SEMANTICS

This specification consists of two parts. First we interpret every query as an assertion about a virtual data base consisting of three distinguishable parts: the given instantaneous data base, the intended resulting data base and temporary data. Such interpretation is "true," "false" or "undefined". Then we define the result of the application of a query to an input data base to be the result-part of a minimal virtual data base satisfying the query.

The embedding of the given data base, of a result data base, and of temporary data in every virtual data base is accomplished by adding the labeling suffixes *-given* and *-result* to the names of the relations. Thus, the "given part" of $vdb \in IDB$ is:

$$\{(\alpha\ r\ \beta) \in Objects \times Relationnames \times Objects |$$
$$(\alpha \text{ append}(r, \text{'}\textit{-given}\text{'})\ \beta) \in vdb\ \}$$

The "result part" is:

$$\{(\alpha\ r\ \beta) \in Objects \times Relationnames \times Objects |$$
$$(\alpha \text{ append}(r, \text{'}\textit{-result}\text{'})\ \beta) \in vdb\ \}$$

I. Interpreting a query q as an assertion about a virtual data base $vdb \in IDB$ with respect to a given data base $idb \in IDB$:

   (a) If the given part of *vdb* is not the given data base *idb,* then the interpretation is *false*. Otherwise proceed.

   (b) Resolve the quantification defining a finite range for every quantifier as follows:

   $D' = \{d \in D \,|\, d$ appears in a fact in $vdb\}$

   Here "d appears in a fact in vdb" means that there is <a,r,b> in vdb such d is a,r, or b. Every "∀" should be resolved as a conjunction of clauses, as follows:

   $(\forall x\ P(x)) \equiv \bigwedge \{P(x) \mid x \in D'\}.$

In the resolved formula there are no more variables.

(c) Interpret the constants as objects of *D*.

(d) Interpret the function symbols.

The semantics of every function symbol is a function from $(D \cup \{undefined\})^2$ to $(D \cup \{undefined\})$

(e) The interpretation of an application of a function symbol to its arguments (which may be expressions themselves) is $f\,(va_1,va_2)$, where f is the semantics of the function symbol, va_1 is the semantics of the first argument ($\in D \cup \{undefined\}$), and va_2 is the semantics second argument, if it appears, and is *undefined* otherwise.

Note: When there is no intuitively meaningful result for a certain function for certain values, the function value need not be *undefined.* Instead it may be 'error' $\in D$. This can be meaningful in intermediate computations within one expression and can be a result value in the *result part* of the data base.

(f) Interpretation of atomic formulae. There are two predicate symbols:

IS-TRUE is interpreted as *true, false* or *undefined* when its argument is interpreted as 'true', 'false' or anything else, respectively.

Is-a-Relationship is interpreted as follows. Let $r, a_1, \ldots, a_n$ be the interpretations of the arguments of this predicate symbol. If any of them equals "*undefined*", then the interpretation of the atomic formula is *undefined.* Otherwise, if the tuple $(r, a_1, \ldots, a_n)$ belongs to the virtual data base, then the interpretation is *true,* else it is *false.*

(g) Interpretation of non-atomic formulae. These are composed of other formulae by "parallel implication" whose three-valued truth table is:

P ⊃ Q	True	False	Undefined
true	true	false	undefined
false	true	true	true
undefined	true	undefined	undefined

Other logical operators are defined as abbreviations:

$\neg x \equiv$ IsTrue(*'false'*)$\supset x$

$x \vee y \equiv (\neg x \supset y)$

$x \wedge y \equiv \neg(\neg x \vee \neg y)$

II. Semantics of a *query* $q \in L$ for a given *idb* $\in$ *IDB*.

Definition. A *vdb* $\in$ *IDB* is called a minimal virtual data base satisfying $q \in L$ with respect to *idb* $\in$ *IDB* if the *assertion* q yields *true* for *vdb* (with respect to *idb*) and yields *false* for every proper subset of *vdb*.

Note that there may be 0, 1 or many such minimal virtual data bases. For the case where there are many, we fix a choice function (which is specified in the proof of the implementability theorem and is dependent on the implementation machinery).

Definition. Let $q \in L$ be a query and *idb* $\in$ *IDB* be a given instantaneous data base. The semantics of q for *idb*, $S_L(q, idb)$, is the "result part" of the chosen (by the fixed choice function) minimal virtual data base *vdb* $\in$ *IDB* satisfying q with respect to *idb*, provided such *vdb* exists; if such *vdb* does not exist, then $S_L(q, idb)$ is *undefined.*

Note: The intuitive meaning of $S_L(q, idb)$ being undefined is the looping of the implementing software.

References

[1] E. F. Codd ''Relational Completeness of Data Base Sublanguages'' in *Data Base Systems* (ed. Rustin). Prentice-Hall, Englewood Cliff, N.J. 1972

[2] F. Bancilhon ''On the completeness of query languages for relational databases.'' Proc. Seventh Symp. on Mathematical Foundations of Computer Science. Springer-Verlag 1978.

[3] E. Codd. ''A Relational Model for Large Shared Data Banks.'' *Communications of ACM*, 13:6.

[4] J.R. Abrial, ''Data Semantics'', in J.W. Klimbie and K.L. Koffeman (eds.), *Data Base Management*, North Holland, 1974.

[5] N. Rishe. *Database Design Fundamentals: A Structured Introduction to Databases and a Structured Database Design Methodology.* Prentice-Hall, Englewood Cliffs, NJ, 1988. 436 pp.

[6] N. Rishe. *Database Design: The Semantic Modeling Approach.* McGraw-Hill, 1992, 528 pp.

[7] A.V. Aho, J.D. Ullman, ''Universality of Data Retrieval Languages.'' Proc. 6th ACM Symp. on Principles of Programming Languages, 1979.

[8] H. Gallaire and J. Minker, eds. *Logic and Data Bases.* Plenum Press, New York, 1978.

[9] H. Gallaire and J. Minker, eds. *Advances in Data Base Theory*, Plenum Press, New York, 1981.

[10] Deyi Li. *A Prolog Database System.* Research Studies Press Ltd, John Wiley & Sons Inc, Letchworth, Hertfordshire, England. 1984.

[11] A.K. Chandra and D. Harel, ''Horn Clauses and the Fixpoint Query Hierarchy.'' Proceedings of the ACM Symposium on Principles of Database Systems. 1982.

[12] A.K. Chandra and D. Harel, ''Computable Queries for Relational Data Bases.'' J. of Computer and System Sciences, vol. 21, 1980.

[13] N. Rishe, ''Database Semantics.'' Technical report TR94-15, School of Computer Science, Florida International University, 1994.