86-PS

›osium on Programming. Proceedings.
nd M. Paul. VI, 262 pages. 1984.
lor Computer Integrated Manufacturing
. Rembold. XVI, 528 pages. 1984.
ıck Shift Registers. II, 1-2, 145 pages.
ıal Conference on Automated Deduction.
508 pages. 1984.
›s: Decision Problems and Complexity.
l by E. Börger, G. Hasenjaeger and
1984.
ıages and Programming. Proceedings.
ıs. VIII, 527 pages. 1984.
a Types. Proceedings, 1984. Edited by
nd G. Plotkin. VI, 391 pages. 1984.
Proceedings, 1984. Edited by J. Fitch.
tions and Boolean Matrix Factorization.
ındations of Computer Science 1984.
by M.P. Chytil and V. Koubek. XI, 581
guages and Their Definition. Edited by
es. 1984.
tive Ergonomics – Mind and Computers.
ıy G.C. van der Veer, M.J. Tauber, T.R.G.
i pages. 1984.
ıply Matrices Faster. XI, 212 pages. 1984.
s Interfaces. Proceedings, 1983. Edited
ıes. 1984.
Software Technology and Theoretical
›dings, 1984. Edited by M. Joseph and
pages. 1984.
ınual Symposium on Theoretical Aspects
ceedings, 1985. Edited by K. Mehlhorn.
ect CIP. Volume I: The Wide Spectrum
P Language Group. XI, 275 pages. 1985.
rks: An Advanced Course. Proceedings,
on, J. Mariani and D. Shepherd. VIII, 497
ndations of Software Development. Pro-
Colloquium on Trees in Algebra and
Edited by H. Ehrig, C. Floyd, M. Nivat and
1985.
ınd Software Development. Proceedings.
n on Software Engineering (CSE). Edited
ıat and J. Thatcher. XIV, 455 pages. 1985.
. Time Series Package (TSPACK). III,
tri Nets 1984. Edited by G. Rozenberg
Genrich and G. Roucairol. VII, 467 pages.
aragon. XI, 376 pages. 1985.
Ansart, G. Hommel, L. Lamport, B. Liskov,
hneider. Distributed Systems. Edited by
VI, 573 pages. 1985.
vey of Verification Techniques for Parallel
985.

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

239

Mathematical Foundations of Programming Semantics

International Conference
Manhattan, Kansas, April 11–12, 1985
Proceedings

Edited by Austin Melton

Springer-Verlag
Berlin Heidelberg New York London Paris Tokyo

POSTCONDITIONAL SEMANTICS OF DATA BASE QUERIES

Naphtali Rishe

Department of Computer Science
University of California
Santa Barbara, CA 93106

A data-independent fully non-procedural language model for binary and relational data bases is designed, in which *all* partial Turing-computable queries are specifiable. A large class of the queries is expressible in a natural and user-friendly way. Every query is formulated in the language as an applied first-order predicate calculus assertion expressing the desired relationship between the state of the data base, the information needed to be displayed, and auxiliary concepts. Interpretation of a query is a partial Turing-computable non-deterministic transformation which for any input state of the data base gives a minimal output to satisfy the assertion.

This general model is implementable effectively but not efficiently. It is intended to serve in investigation, generation and extraction of sublanguages which are user friendly and efficiently implementable.

The proposed general language has sublanguages which are intended to restrict use of undesirable or meaningless operations on objects. One of the important cases differentiates between abstract objects, representing real-world entities, and concrete values. A more general case is parametrized by a family of permitted operations on the domain of objects and its subdomains. The sublanguages are able to express every data base transformation reasonable within the restrictions parametrizing the sublanguages.

1. MOTIVATION

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 called 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.

Other data base languages are called 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 interpretation of queries, interpretation of updates usually also depends on some laws fixed for an application. These laws are known as integrity and inference laws, 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 [Bancilhon-78]:

- 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 [Codd-70] (where an instantaneous data base is a collection of named n-ary *mathematical relations* over domains of values) or a binary data model — [Abrial-74] (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 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 requirements are sometimes restricted to those expressible by Codd's Relational Algebra (or by Relational Calculus), and thus languages whose expressivity is equivalent to Relational Calculus are called Codd-complete (*cf.* [Codd-72]). [Aho and Ullman - 79] showed that quite reasonable queries, such as those involving transitive closure are unexpressible in Codd's Algebra and proposed to extend the Algebra by a fix-point operator. A more powerful class of languages, using Horn clauses, is advocated in [Gallaire-78] and [Gallaire-81]. A representative of this class is Prolog (*cf.* [Li-84]). Incomplete expressibility of Horn clauses was shown in [Chandra and Harel - 82]. [Chandra and Harel - 80] propose 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 necessitates interpretation of values. (Their language has a powerful capability of calculation of values, including aggregative calculations, e.g. counting, but does not allow all meaningful computations.) Unfortunately, their language is highly-procedural, unnatural and inconvenient to use. 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 laws.

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

(1) It is **absolutely complete**, 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.)

(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 binary-oriented and the table-oriented user are provided with appropriate syntax.
- The user can easily specify calculations on values when needed. Arbitrary aggregative 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. (More "sugar" is still desired.)
- The same syntax can be used to specify update transactions and integrity and inference laws.

(3) The language is implementable. Yet, heuristic techniques need to be designed to implement efficiently some important subsets of the language. Otherwise the language will serve principally as a model for generation of efficiently-implementable *sub*languages and for comparison of different languages.

(4) For any definition of data-independency 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.

2. 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*, 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 **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;
- **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 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 data base 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 principle variant of the language, tri-valued parallel logic is used.)
- function symbols expressing scalar mathematical operations over the domain of objects, including comparators (>,<, etc.) yielding Boolean values.

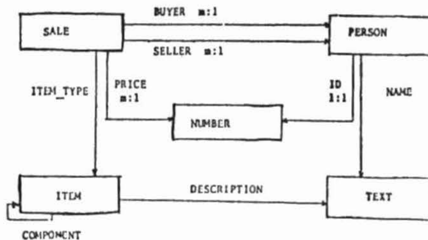
Two alternative variants of the language have been developed. The first one, focuses the assertional semantics on data base structure, while the scalar mathematical operations are expressed, for the sake of separation from information-manipulation, by an infinite set of function symbols syntaxed as recursive 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 ([Rishe-85]), 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 "=", *inter alia* 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 unexpressible by Horn clauses, e.g. Example 3 of the next section.

Sublanguages have been investigated where some function symbols are used, while other 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 [Rishe-85]. Among these "sugar" abbreviations are: a full scope of logic operators; contextual defaults for quantifiers; abbreviations of sub-assertions expressing aggregative application of associative scalar functions, *e.g.*, summation, counting etc; substitution of variables by *examples* of objects (inspired by Zloof's *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.

3. EXAMPLES OF QUERIES

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



1) 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 ((x GOT-A-BOLT*result*)).

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'

2) 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]

3) 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 is the DESCRIPTION of w and
result:
      watch HAS-MANY-COMPONENTS and
if stone (e.g.) is PAIRED-TO wheel (e.g.) and
      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.
            given:
                  stone is PAIRED-TO wheel.
```

4) /* 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)[1] of PRICE of
          bargain dependent on man s.t. (man is the SELLER
          of the bargain)) and
        result: the INCOME of smith is income.
```

4. REVIEW OF MAIN THEOREMS ABOUT THE LANGUAGE

The following is a review of main results about the proposed language model. They are proven in [Rishe-85]. The proofs of the most important results are outlined in the appendices of this paper.

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

[1] This phrase, which might look aggregative and second order, is a syntactic sugar abbreviation for a longer *first-order non-aggregative* phrase using only one binary function symbol "+" which is applied to pairs of integers denoting prices.

2) The language is ***absolutely* complete,** i.e. for every partial computable function ϕ: $IDB \rightarrow IDB$ 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 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 syntacticly 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 restrictions is expressible in the restricted language up to an isomorphism.

   More precisely: for every set Φ of functions over the domain of objects, for every computable (Φ, C)-preserving data base transformation ϕ, there exists a query q, using no other functions symbols or constants but Φ and C, whose semantics is $(\Phi \cup C)$-isomorphic to ϕ.

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

8) The language can be used to specify every **integrity** and **inference** law 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 laws.

5. FORMAL SEMANTICS

$IDB \equiv$ *INSTANTANEOUS-DATA-BASES*[2]

ASSERTION:

[2] For the relational model (having a denumerable domain D of objects),

$$IDB \equiv \text{POWERSET}(NAMES-OF-RELATIONS \times D^{*})$$

Using the binary model, $D \cup D \times D$ is sufficient instead of D^{*}.

$[\mathit{THE\text{-}LANGUAGE} \rightarrow [\mathit{IDB} \rightarrow \mathit{BOOLEAN}_\perp]]$[3]

semantics-of-queries:

$[\mathit{THE\text{-}LANGUAGE} \rightarrow [\mathit{IDB} \rightarrow \mathit{OUTPUT}_\perp]]$

semantics-of-data-manipulation:

$[\mathit{DATA\text{-}SEMANTIC\text{-}FUNCTIONS} \rightarrow [\mathit{THE\text{-}LANGUAGE} \rightarrow [\mathit{IDB} \rightarrow \mathit{IDB}_\perp \times \mathit{OUTPUT}_\perp]]]$

DATA-SEMANTIC-FUNCTIONS=

$[\mathit{IDB} \rightarrow \mathit{IDB}_{\perp,\mathit{ERROR}}]$

semantics-of-laws:

$[\mathit{THE\text{-}LANGUAGE} \times \mathit{THE\text{-}LANGUAGE} \rightarrow \mathit{DATA\text{-}SEMANTIC\text{-}FUNCTIONS}]$

$$\textbf{semantics-of-queries}(\underset{\mathit{THE\ LANGUAGE}}{\underset{\in}{query}}, \underset{\mathit{INSTANTANEOUS\ DATA\ BASES}}{\underset{\in}{idb}}) =$$

let $\phi = \mathit{ASSERTION}(query)$ in

$$\text{choose}_\perp\left(\left\{ (result, temp) \;\middle|\; \begin{array}{c} \phi(idb \dot\cup result \dot\cup temp) \wedge \\ \forall vdb. \\ \text{if } idb \subseteq vdb \subset (idb \dot\cup result \dot\cup temp) \\ \text{then } \phi(vdb) \equiv false \end{array} \right\}\right)_{\substack{\text{project1} \\ \text{(take only } result \text{ component)}}}$$

[4]

$$\mathit{ASSERTION}_{query}(\underset{\mathit{INSTANTANEOUS\ DATA\ BASES}}{\underset{\in}{idb}}) =$$

case *query*:

$$\begin{bmatrix} \forall variable: q & \bigwedge_{x \text{ in } idb} \mathit{ASSERTION}_{q\ldots}(idb) \\ q_1 \supset q_2 & \mathit{ASSERTION}_{q_1}(idb) \supset \mathit{ASSERTION}_{q_2}(idb)^{6} \\ \text{IsaRelationship}(expression_1, \ldots) & (\mathit{Evaluate}(expression_1), \ldots) \in idb \\ \text{IsTrue}(expression) & \mathit{Evaluate}(expression) \underset{\perp}{=} \underline{true} \end{bmatrix}$$

---

[3] $X_\perp$ is $X \cup \{\perp\}$; $\perp$ means *undefined*.

[4] $\dot\cup$ is a decomposable union: for $\alpha,\beta,\gamma,\delta \in IDB$, $\delta = (\alpha \dot\cup \beta \dot\cup \gamma)$ **iff** $(\alpha,\beta,\gamma) = \text{split}(\delta)$

[6] $\supset$ has a parallel logic tri-valued interpretation: $(false \supset ?) \equiv (? \supset true) \equiv true$; $(true \supset false) \equiv false$. In the rest of the cases it is $\perp$ (The other operators, e.g. $\wedge, \vee$, and $\sim$, are syntactic abbreviations using $\supset$.)

ABSTRACTED SYNTAX

For any given decidable set D of all possible objects, for any given decidable set Φ of partial computable functions from D^* to D, $\mathit{THE\text{-}LANGUAGE}_{D,\Phi}$ is defined by:

$$\textbf{assertion} ::= \begin{bmatrix} \forall variable\ assertion \mid^{variable}_{constant} \\ assertion \supset assertion \\ \text{IsaRelationship}(expression^*)^{7} \\ \text{IsTrue}(expression) \end{bmatrix}$$

expression: $\mathit{EXPRESSIONS}_{D,\Phi} = D \cup \Phi \times \mathit{EXPRESSIONS}^*_{D,\Phi}$

variable: *VARIABLES* – a denumerable set

$\exists$, $\wedge$, $\vee$, $\neg$, TRUE, FALSE are standard abbreviations using $\forall$, $\supset$, IsTrue(*false*).

SPECIAL CASES

1) First Order: Relation Names May Not Be Quantified.

The head of any tuple in "IsaRelationship" is not quantified and stands for a name of a relation.

2) The *Uninterpreted Domain* case.

$$\Phi = \{=\}, \quad (= : D \times D \rightarrow \underset{D}{\underset{\cap}{BOOLEAN}})$$

$$\textbf{assertion} ::= \begin{bmatrix} \forall variable\ assertion \mid^{variable}_{constant} \\ assertion \supset assertion \\ \text{IsaRelationship}(constant^*) \\ constant = constant \end{bmatrix}$$

constant: D

variable: *VARIABLES* – a denumerable set

---

[7] $expression^*$ is a tuple. Triples representing binary relationships are sufficient to consider since higher-order relations are derivable.

3) An ***Uninterpreted Subdomain*** **and a** ***Fully Interpreted Subdomain***.

$D=U \cup I;$

$I=INTEGERS$

$\Phi=\{=\} \cup [I^{*} \underset{computable}{\overset{partial}{\longrightarrow}} I]$

APPENDICES

Appendix 1 - The Implementabilty Theorem

The proof of the 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. 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 tri-valued logic above).

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 base 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 E1 (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 (were 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 ϕ: $IDB \rightarrow IDB$ 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 for an appropriate query for the language L' not using variables or expressions as names of relations (to be used in "L' 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 $f = tr \cdot \phi \cdot tr^{-1}$. 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:

$\varnothing$*-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 $\perp$).
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: $IDB \xrightarrow{P} IDB$ 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: $IDB \xrightarrow{P} IDB$ has a *finite* set of relevant relation-names iff there is a finite set which contains all the relation-names relevant for ϕ.

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

Theorem. The language L' (i.e., those queries of L which use only constants as names of relation) 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 $\overline{q}_2$ be the semantics of q_2. I shall show that $\overline{q}' = \phi$.

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 $\overline{q}'(idb) \equiv \phi(idb - D \times (D-S) \times S)$ which in turn, by the condition of the theorem and the definitions above, is equivalent to $\phi(idb)$.

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

Appendix 3: Language Submodels with Restrictions of Value Calculation; Isomorphism of Queries.

The purpose of this appendix is to clarify the result 6 in section 4.

Finite set of basic functions without recursion can be 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 the following I wish to actually restrict the power of the language by removing from it the ability to specify computations which are extremely unnatural and should be forbidden in a user's system of concepts. The general case needed to be investigated is the one in which a user is provided with a family of functions on values considered legal 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 power of first-order predicate calculus. Usually no computation on abstract objects in the binary model of data bases may be regarded meaningful.

Families 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 value, 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. (Instead of program control power, a first-order predicate calculus can be used.)

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

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

Though binary operations are sufficient to have the complete power of the language, I am aiming to restrict the power and to be able to model exactly any practical restriction. That's why I 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 involve computation of any values, but Φ-basable values, and does not condition data base structure on such values.

These claims and the following ones will be respecified 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 obsolete, i.e. the use of 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 ϕ: $IDB \xrightarrow{\bullet} IDB$ 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.

REFERENCES

[Abrial - 74] J.R. Abrial, "Data Semantics", in J.W. Klimbie and K.L. Koffeman (eds.), *Data Base Management*, North Holland, 1974.

[Aho and Ullman-79] A.V. Aho, J.D. Ullman, "Universality of Data Retrieval Languages", in *Proc. 6th ACM Symp. on Principles of Programming Languages*, 1979.

[Bancilhon - 78] F. Bancilhon "On the completeness of query languages for relational databases". Proc. Seventh Symp. on Mathematical Foundations of Computer Science. Springer-Verlag 1978.

[Codd - 72] E. F. Codd "Relational Completeness of Data Base Sublanguages" in *Data Base Systems* (ed. Rustin). Prentice-Hall, Englewood Cliff, N. J. 1972

[Codd - 70] E. F. Codd. "A Relational Model for Large Shared Data Banks." CACM . v. 13 n. 6. 1970.

[Chandra and Harel-80] A.K. Chandra and D. Harel, "Computable Queries for Relational Data Bases", J. of Computer and System Sciences, vol. 21, 1980.

[Chandra and Harel -82] A.K. Chandra and D. Harel, "Horn Clauses and the Fixpoint Query Hierarchy", Proceedings of the ACM Symposium on Principles of Database Systems. 1982.

[Gallaire-78] H. Gallaire and J. Minker, eds. *Logic and Data Bases*. Plenum Press, New York, 1978.

[Gallaire-81] H. Gallaire and J. Minker, eds. *Advances in Data Base Theory*, Plenum Press, New York, 1981.

[Li - 84] Deyi Li. *A Prolog Database System*. Research Studies Press Ltd, John Wiley & Sons Inc, Letchworth, Hertfordshire, England. 1984.

[Rishe-85] N. Rishe, *Semantics of Universal Languages and Informations Structures in Data Bases*. Technical report TRCS85-010, Computer Science Department, University of California, Santa Barbara, 1985.