Prolog and Relational Databases for Fifth Generation Computer Systems

by

Susumu Kunifuji

Haruo Yokota

PROLOG and Relational Data Bases

for Fifth Generation Computer Systems

by

Susumu Kunifuji
Haruo Yokota

Research Center
Institute for New Generation Computer Technology (ICOT)

Mitakokusai Building (21F)

1-4-28, Mita, Minato-ku, Tokyo, 108, Japan

ABSTRACT

Among several research subjects on the Fifth Generation Computer Systems Project in Japan, there are two major research plans, that is, design and implementation of a Sequential Inference Machine and a Relational Data Base Machine. The zero-th version of the Fifth Generation Kernel Language is based on the logic programming language PROLOG with several functions added. One of these functions is the interface between the Sequential Inference Machine and the Relational Data Base Machine.

The authors' research theme on the Fifth Generation Computer Systems is the application of logic programming to relational data bases. First, the authors consider the question of how powerful a relational database query language should be. They show that Prolog with a meta-predicate *setof* is a relationally complete query language, and that it can handle the concepts corresponding to least fixed point operators. Second, they point out that there exist two approaches to interface a PROLOG-like logic programming system with large relational databases; which are the so-called evaluational and non-evaluational approach. Current implementation of PROLOG assumes that the relational database resides in an Internal Data Base (primary memory). However, when the relational database resides in an External Data Base (direct accessible secondary memory), several difficut "interface" problems appear and the utility of PROLOG is reduced. This is because the Sequential Inference Machine is designed and implemented by the non-evaluational approach and the Relational Data Base Machine is done by the evaluational approach. Therefore, the authors describe how to combine these machines and propose a simple way in which the PROLOG-like system can be modified to operate efficiency with large relational databases. In other words, they show how a PROLOG system can be adapated for the evaluational type. They give,

with respect to the non-evaluational type, a method which modifies PROLOG in a simple manner.

In conclusion, Fifth Generation Kernel Language logic programming can be used to express and solve many fundamental and difficult problems regarding deductive question-answering on relational databases.

1. Introduction

The Fifth Generation Computer Systems (FGCS) Project in Japan [1], [2] has been planned and inaugurated in order to bring about new development in information processing and computer industries in the 1990's. The research and development (R&D) targets of the FGCS are such core functions for the Knowledge Information Processing Systems (KIPS) in the 1990's as problem-solving and inference systems, knowledge base management systems, and intelligent man-machine interface systems. The R & D period set for this project is 10 years, divided, as shown in Fig. 1 [1], into an initial stage (3 years), a middle stage (4 years), and a final stage (3 years).

For the initial stage, the most important R & D targets concerning the problem-solving and inference systems and the knowledge base management systems are the design and implementation of the Sequential Inference Machine (SIM) and the Relational Data Base Machine (RDBM) respectively. The goal of the SIM is to develop a high-performance firmware baced machine which efficiently supports the specification of the Fifth Generation Kernel Language (FGKL), which will become the core logic programming language of all software systems, and includes various functions to provide researchers with a good programming environment to accelerate the R & D of software systems. The goal of the RDBM is the R & D of a high-level architecture to support Relational Data Base (RDB) management functions to form part of the core of the FGCS, and capable of storing, quickly retrieving, and efficiently updating very large RDBs. For this goal, the R & D plan will include the design and experimental implementation of the RDBM and the RDBMS (RDB Management System).

Toward the aim of the FGCS, most essential is an enhancement of abilities for problem-solving and inference. One of the major reasons for the adoption of the logic programming language is that it includes a basic inference mechanism. PROLOG provides a natural and useful logic programming language to represent RDB, Knowledge Base (KB), and inference mechanism and to solve various problems connected with them. Therefore, the FGKL is based on PROLOG with several functions added. Among these functions, the most interesting problem is how to combine the SIM and the RDBM and how to provide a simple way in which the SIM can be modified to interface efficiency with the RDBM.

2. Deductive Power in a Relational Database Query Language

2.1 Basic Approaches

Our main concern of how to represent knowledge in logic is based upon the recognition that a language of logic can be used to represent RDBs, KBs, and inference mechanisms [3] and consequently that it provides a uniformity of knowledge representation and hence a firm mathematical basis for knowledge engineers [4].

It has recently been recognized by Gallaire et al. [3] that predicate calculus is very useful for deductive question-answering on RDBs. The methods presented by many authors show how far predicate-calculus can be used to express and to solve many fundamental and difficult problems regarding deductive question-answering systems, and to understand the implications to RDBs.

In the research of deductive question-answering systems on RDBs, there exist two kinds of approaches from a logical point of view, one of which is called a non-evaluational approach and the other is called an evaluational approach [5].

In the non-evaluational approach, we regard a logic as a formal system which consists of axioms and inference rules, and formulate the formal aspects of RDBs and KBs by appealing to the same set of axioms of the formal system. This approach admits an application of inference rules and fits the current implementation of PROLOG, which assumes that the RDB resides in primary memory, which is called the Internal Data Base (IDB). However, it does not admit any use of set-operation mechanisms of relational algebra offered by our experimental implementation of the RDBM/the RDBMS.

In the evaluational approach, we regard a logic as a language which consists of syntax and semantics. We interprete an RDB as a model given in its semantics, and a KB as the semantic constraints of the model. Under this approach, we can use the above-mentioned set-operation mechanisms of relational algebra. Accordingly, with the development of RDBMSs, the advantage of this approach has recently been recognized. In spite of the advantage, however, such specific interpretation techniques as connection graphs [6] and rewriting-rules [5] have not enough been developed. Therefore, this approach fits the current implementation of the RDBM/RDBMS, which assumes that RDB resides in secondary memory, which is called to the External Data Base (EDB), but it does not fit the current implementation of PROLOG.

We have been discussing the advantages and disadvantages of these two approaches. We are concerned about interface hardware and software between the SIM and the RDBM. This paper gives a simple method to integrate the non-evaluational approach and the evaluational one, after the knowledge representative power of the PROLOG-like logic programming language has been investigated.

2.2 Basic Concepts

In this section, we will give some preliminary remarks on notation and terminology.

We will consider a formal system which consists of a first-order Horn logic. In general, an expression is any sequence of symbols. There are three kinds of meaningful expressions: term(s), literal(s), and Horn clause(s).

A term is an individual constant, an individual variable, or an expression of the form $f(t_1, \cdots, t_m)$, where f is an m-ary function constant and $t_1, \cdots t_m$ are terms.

A literal is either a positive atomic formula or a negative atomic formula. An atomic formula is an expression of the form $p(t_1, \cdots, t_n)$, where p is an n-ary predicate constant and $t_1, \cdots, t_n$ are terms.

A Horn clause is a set of literals of the form

$$(\forall x_1) \cdots (\forall x_k)(L_0 \vee L_1 \vee \cdots \vee L_\ell),$$

where L_i's are literals, at most one of which is positive. Note that it is a kind of a closed universally quantified well-formed formula. If L_0 is the only positive atomic formula, we will write the above-mentioned Horn clause in the DEC-10 PROLOG [7] form

$$L_0 :- L_1, L_2, \cdots, L_\ell.$$

L_0 is called the conclusion of the clause, and L_i's are called its conditions or its body. A Horn clause containing no positive atomic formula is called a question or a goal statement and written in the form

$$?- L_1, L_2, \cdots, L_\ell.$$

A Horn clause containing only the positive atomic formula is called an assertion or fact and written in the form

$$L_0.$$

2.3 Relationally Complete Query Language

After Codd [8] gave the concept of relational completeness, Codd's relational algebra is often used as a model of an RDB query language. In this section, we show that DEC-10 PROLOG with a meta-predicate *setof* [9] (see appendix (A)) is a relationally complete query language.

(Proof of relational completeness)

An RDB can be built up as a set of assertions in an IDB of the PROLOG.

A. For set operations:

The concepts of intersection, union, difference, and Cartesian product can be respectively implemented in the PROLOG as follows:

(1) *setof*($(X_1, \cdots, X_n)$, $(p(X_1, \cdots, X_n), q(X_1, \cdots, X_n))$, S),

(2) *setof*($(X_1, \cdots, X_n)$, $(p(X_1, \cdots, X_n); q(X_1, \cdots, X_n))$, S),

(3) *setof*($(X_1, \cdots, X_n)$, $(p(X_1, \cdots, X_n), not(q(X_1, \cdots, X_n)))$, S),

(4) *setof*($(X_1, \cdots, X_n, Y_1, \cdots, Y_m)$, $(p(X_1, \cdots, X_n), q(Y_1, \cdots, Y_m))$, S).

B. For relational algebra operations:

The concepts of projection, θ-join, θ-restriction, and division can be respectively implemented in the PROLOG as follows:

(5) *setof*($(X_{i_1}, X_{i_2}, \cdots, X_{i_m})$, $(X_{j_1}, \cdots, X_{j_{n-m}})\hat{}p(X_1, \cdots, X_n)$, S),

(6) *setof*($(X_1, \cdots, X_n, Y_1, \cdots, Y_m)$, $(X_i \theta X_j, p(X_1, \cdots, X_i, \cdots, X_n),$

$q(Y_1, \cdots, Y_j, \cdots, Y_m))$, S),

(7) *setof*($(X_1, \cdots, X_i, \cdots, X_j, \cdots, X_n)$, $(X_i \theta X_j,$

$p(X_1, \cdots, X_i, \cdots, X_j, \cdots, X_n))$, S),

(8) *setof*($(X_{k_1}, \cdots, X_{k_{n-\ell}})$, $(setof((X_{i_1}, \cdots, X_{i_\ell}), p(X_1, \cdots, X_n), S_p)$

setof($(Y_{j_1}, \cdots, Y_{j_\ell})$, $(Y_{h_1}, \cdots, Y_{h_{m-\ell}})\hat{}q(Y_1, \cdots, Y_m)$, S_q),

subset(S_q, S_p)), S),

where $X_{i_1} = Y_{j_1}$, $X_{i_2} = Y_{j_2}$, $\cdots$, and $X_{i_\ell} = Y_{j_\ell}$.

From A and B, the relational completeness of PROLOG with the *setof* operators is shown. (q.e.d.)

See appendix (B) which demonstrates some typical examples of the above operations.

Note that the meta-predicate *setof* is not a primitive predicate of any first-order logic, and the 2nd argument of the *setof* predicate is not a term, but a sequence of conditions of any Horn clause. Also, note that the *setof* predicate is implemented by using many higher-order predicates and meta-logical control primitives. (see appendix (A)).

2.4 Least Fixed Point Operator

The characteristic of logic programming in PROLOG on RDBs is easily able to represent and utilize recursive definitions. From this point of view, there is an important class of query languages that cannot be expressed in relationally complete query languages, i.e. relational algebra or relational calculus. Aho et al. [10] considered the question of how powerful a relational query language should be. They stated the universal conditions of the query language. Their conditions are known as a class of the universal query language which can handle a least fixed point operator.

Consider an equation of the form

(9) $R = R \cup g(R)$

where g(R) is a monotone relational algebra expression [10]. A least fixed point of equation (9) is a relation R* such that

(10) $R^* = R^* \cup g(R^*)$ and

(11) If R is any relation such that $R = R \cup g(R)$, then $R^* \subseteq R$.

For the equation (9), they provide a procedure of constructing a relation R* inductively for a given existing base relation R_0 ($\subseteq R$). The procedure can be expressed in terms of a simple PROLOG program as follows:

```
(12)  least-fixed -point (R0, R*):-setq(R0, R), lfp(R, R*).

      lfp(R, R*) :-setq(R, R'), g(R, R''), union (R, R'', R'''),
                   noteq(R', R'''), !, lfp(R''', R*).

      lfp(R, R):-!.
```

where the predicate setq(X, Y) means the assignement statement "X ← Y." The above PROLOG program is not a complete description which would require the detailed definition of the low-level predicates. Note that this procedure essentially contains a tail recursive optimization problem of PROLOG [11].

At first sight, the problem of the above-mentioned least fixed point operators is seemed quite difficult to solve. However, its solution is

simple, because we can also use the meta-predicate *setof* to solve it. We illustrate with an example of how to PROLOG handles this more general class of the query language.

[Example 1]

(13) ancestor (X, Y):- parent (X, Y).

ancestor (X, Y):- parent (X,Z), ancestor (Z, Y).

parent (x_1, y_1).

⋮

parent (x_n , y_n).

(14) ?- *setof*((X, Y), ancestor (X, Y), Set-of-Ancestor).

The problem of this approach is how to find a logical form of the least fixed point equation (9) and how to express it in terms of a"right" recursive Horn clause in PROLOG. Note that the problem is a problem of PROLOG programming itself, and not of implementation of the least fixed point operators.

3. Interface between PROLOG and RDBMS

3.1 *replace* PROLOG Program

In this section, we propose how to integrate PROLOG on the SIM and RDBMS on the RDBM. Some implemental modifications should be made to the PROLOG program to permit a lazy evaluation of a sequence of conditions which may be handled by a RDBMS. The basic idea is the same as Bowen et al. [12] and Chakravarthy et al. [14]. The former paper gives a theoretical basis of amalgamating object-language and meta-language in logic programming, but their '*demo*' program is not implemented in PROLOG so far as the authors know. The latter paper gives a PROLOG-based inplementation image, but their '*demo*' program cannot handle recursive Horn clauses, negative literals, and other evaluable predicates. Our '*replace*' PROLOG program amalgamates object-language and meta-language to avoid modifications to the PROLOG complier/interpreter, and moreover it can handle general recursive Horn clauses, negative literals, and other predicates that can be evaluated. But it cannot manage a cut operator '!' [7] which is not essential in RDBMS.

The basic idea of the '*replace*' program's algorithm is as follows:

(i) A new evaluable predicate called '*edb*' is introduced for each EDB relation which is a so-called base relation managed by RDBMS.

(ii) *edb* predicates incorporated into new conditions are placed at the end of a sequence of given conditions. On the other hand, other predicates are added at the front of old conditions.

(iii) A condition which starts with an *edf* predicate should be recognized as a lazy evaluation state.

(iv) If all conditions are recognized as lazy evaluation states, a query for a given sequence of conditions must be sent to a RDBMS to obtain one answer. Other alternative queries can be message-passed by usual backtracking mode of the PROLOG, i.e. typing alternative command ';'.

The following PROLOG program answers the above-mentioned problem. It can be regarded as a top level interpreter of a Horn clause program which represents Horn clause provability [12].

```
/* Replace prolog prog. for EDB */

replace(_,[],[]) :- !.
replace(_,[Head|Body],[Head|Body]) :-
        Head =.. [edb|Rest], !.
replace(Prog,[Goal|Rest],AllReplace) :-
        prog(Clause,Prog),
        rename_vars(Clause,[Goal|Rest],[Head|Body]),
        differ(Goal,Head,Diff),
        add(Prog,Body,Rest,InterGoals),
        apply(InterGoals,Diff,NewGoals),
        replace(Prog,NewGoals,AllReplace).
```

The first argument of the *replace* predicate indicates the target program, the second one gives the source goal statement, and the last one receives the replaced goal statement which consists entirely of *edb* predicates.

The first clause of the *replace* predicate states that recursion is stopped when the program achieves an empty collection of goals. In the second clause of the predicate, Head =..[edb|Rest] means that Head consists of an *edb* predicate. It indicates that the goal statement consists only of an *edb* predicate, and then recursion should be stopped.

The last clause of the *replace* predicate states:

1. finding a clause in the target program indicated by 'Prog',

2. renaming the variables in the clause so they are distinct from the variables in the goal statement,

3. matching the top of the goal statement list with the head of the clause, and assigning the difference of the variables between the top of the goal statement list and the head of the clause to 'Diff',

4. adding the body of the clause to the rest of the goal statement; and at this time, if the body contains only an *edb* predicate, then adding it to the end of the list; otherwise, adding it to the top of the list,

5. applying the matching substitution to obtain a new collection of goals,

6. replacing the new goal statement in the same way (recursively).

A complete version of the *replace* PROLOG program is given by appendix (C).

3.2 Examples

In this section, we illustrate the execution of the above program with the following typical examples:

[Example 2]

Consider the following program (15) and (16) and the goal (17) stored in IDB.

(15) fallible (X):- human (X).

(16) human (turing).
human (socrates).
greek (socrates).

(17) ?- fallible (X), greek (X).

In the evaluational approach, the assertion-type clauses are stored in EDB as RDBs. Therefore, we replace all assertion-type ones by the new Horn clauses in IDB corresponding to the RDBs in EDB.

(18) human (X):- edb (human, X).
greek (X):- edb (greek, x).

Then logic programs (15) and (16) are transformed to the following form which is an object-language program f_1 (17) in meta-language programs:

(17) prog([human(x),edb(human,x)],f1).
prog([greek(x),edb(greek,x)],f1).
prog([fallible(x),human(x)],f1).

where variables are distinguished by an initial letter x, y, and z only.

The meta-language program executes the goal statement (18) against the object-language program (17), and returns the answers (19).

```
(18) ?- replace(f1,[fallible(x),greek(x)],_45).

(19) _45 = [edb(human,x),edb(greek,x)] ;

     no
```

As the answer "edb (human, x), edb (greek, x)" is evaluable in an RDBMS, it would return all answers which are x = socrates in this case. It is the result of what occurs for only one execution path. Though, generally speaking, all possible execution paths must be executed, the above program (19) shows that there is no other execution path.

[Example 3]

Consider the following "right" recursive program (20):

```
(20) ancestor (X, Y):- parent (X, Y).
     ancestor (X, Y):- parent (X, Z), ancestor (Z, Y).
```

In this case, the program (20) is transformed into the following object-language program f_2 (21) in a meta-language:

```
(21) prog([parent(x,y),edb(parent,x,y)],f2).
     prog([ancestor(x,y),parent(x,y)],f2).
     prog([ancestor(x,y),parent(x,z),ancestor(z,y)],f2).
```

The meta-language *replace* program executes the goal statement

```
(22) ?- replace(f2,[ancestor(x,y)],_53).
```

Then it returns all possible execution paths, if necessary, by typing the alternative command ";", which is supported by the automatic-backtracking mechanism in the PROLOG.

```
(23) _53 = [edb(parent,x,y)] ;

     _53 = [edb(parent,x,z),edb(parent,z,y)] ;

     _53 = [edb(parent,x,z),edb(parent,z,z72),edb(parent,z72,y)]

     yes
```

[Example 4]

Consider the following more complicated logic program (24) and its transformed logic program f3 (25).

(24) $q_1(X, Y)$:- $p_2(X, Z)$, $p_3(Z, Y)$.

$q_1(X, Y)$:- $p_2(X, Z)$, $p_3(Z, Y_1)$, $p_4(Y_1, Y)$.

$q_2(X)$:- $p_1(Y)$, $p_6(Y, X)$.

```
(25) prog([p1(x),edb(p1,x)],f3).
     prog([p2(x,y),edb(p2,x,y)],f3).
     prog([p3(x,y),edb(p3,x,y)],f3).
     prog([p4(x,y,z),edb(p4,x,y,z)],f3).
     prog([p6(x,y),edb(p6,x,y)],f3).
     prog([q1(x,y),p2(x,z),p3(z,y)],f3).
     prog([q1(x,y),p2(x,z),p3(z,y1),p4(x,y1,y)],f3).
     prog([q2(x),p1(y),p6(y,x)],f3).
```

For this program (25), the next question (26) and its answers (27) suggest that the *replace* program also permits a question which may contain some constants. In this case, the question (26) corresponding to "?-q_1(a, X), q_2(X)." contains a constant a.

```
(26) ?- replace(f3,[q1(a,x),q2(x)],_81).

(27) _81 = [edb(p2,a,z),edb(p3,z,x),edb(p1,y),edb(p6,y,x)] ;

     _81 = [edb(p2,a,z),edb(p3,z,y1),edb(p4,a,y1,x),edb(p1,y),edb(p6,y,x)] ;

     no
```

[Example 5]

Consider the following "left and right" recursive program (28):

(28) ancestor (X, Y):- parent (X, Y)
ancestor (X, Y):- ancestor (X, Z), ancestor (Z, Y)

For this program, the usual PROLOG programming generates a looping program which does not stop.

However, against its transformed program (29), if the *replace* program executes a given goal statement (30), then it can return a sequence of backtracking choices (31), if necessary, of a finite length as one wishes.

```
(29) prog([parent(x,y),edb(parent,x,y)],f4).
     prog([ancestor(x,y),parent(x,y)],f4).
     prog([ancestor(x,y),ancestor(x,z),
             ancestor(z,y)],f4).

(30) ?- replace(f4,[ancestor(x,y)],_43).

(31)  _43 = [edb(parent,x,y)] ;

      _43 = [edb(parent,x,z),edb(parent,z,y)] ;

      _43 = [edb(parent,x,z),edb(parent,z,z22),edb(parent,z22,y)]

      yes
```

[Example 6]

Consider the following program (32) which contains an evaluable predicate 'neq' and a negation 'not':

```
(32) manager_many (X, Y):- manager (X, Y), manager (X, Z), neq (Y, Z).

     manager_one (X, Y) :- manager (X, Y), not (manager_many (X, Y)).
```

Note that PROLOG's 'not' is not a logical negation, but a convention for negative information representation in artificial intelligence techniques [3].

For its transformed program (33), the *replace* program is able to execute a given statement (34), and it returns an answer (35), in which 'not' is handled in a nested form and 'neq' is done as an evaluable *edb* predicate.

```
(33) prog([manager(x,y),edb(manager,x,y)],f5).
     prog([neq(x,y),edb(neq,x,y)],f5).
     prog([manager_many(x,y),manager(x,y),
             manager(x,z),neq(y,z)],f5).
     prog([manager_one(x,y),manager(x,y),
             not(manager_many(x,y))],f5).

     prog([not(X),not(X)],F).

(34) ?- replace(f5,[manager_one(x,y)],_43).

(35) _43 = [edb(manager,x,y),
            not(edb(manager,x,y),edb(manager,x,z),edb(neq,y,z))] ;
     no
```

[Example 7]

The most difficult problem in integrating the evaluational approach and the non-evaluational one is how to handle the case where part of the assertion-type clauses resides in the IDB and the rest exists in the EDB. Consider the following program (36) in such a case:

(36) ancestor (X, Y):- parent (X, Y).
ancestor (X, Y):- parent (X, Z), ancestor (Z, Y).
parent (tom, bill).

For its transformed program (37), the *replace* program executes a given goal statement (38), and then the final answers that result from its execution are shown in (39):

(37)
```
prog([ancestor(x,y),parent(x,y)],f6).
prog([ancestor(x,y),parent(x,z),ancestor(z,y)],f6).
prog([parent(tom,bill)],f6).
prog([parent(x,y),edb(parent,x,y)],f6).
```

(38)
```
?- replace(f6,[ancestor(tom,x)],_43).
```

(39)
```
_43 = [] ;

_43 = [edb(parent,tom,x)] ;

_43 = [edb(parent,bill,x)] ;

_43 = [edb(parent,bill,tom)] ;

_43 = [edb(parent,bill,z),edb(parent,z,x)]

yes
```

Careful consideration must be given in order to capture the meaning of the above answers. All possible execution paths are complex in cases where assertion-type clauses exist in IDB as well as EDB.

4. Conclusion

For the initial stage of Japan's FGCS project, the most important R & D targets are the design and implementation of IIM and RDBM. Then we are interested in the research on the interface software between PROLOG on the SIM and RDBMS on the RDBM.

First, we have investigated the knowledge representative power of logic programming in PROLOG. We show that PROLOG is a relationally complete query language and, furthermore a universal relational query language embedding the least fixed point operators. The proof is shown by means of a meta-logical predicate *setof*. The predicate is useful in determining all of the terms that satisfy some condition in various applications, and attaches importance to suggest how to convert relational calculus expressions to relational algebra expressions.

Second, we described two approaches for deductive question-answering on RDBs. In the non-evaluational approach, a theorem prover solves a given problem by making use of IDB's knowledge which consists of assertions and general rules. Though this approach fits the current implementation of the PROLOG, it cannot manage very large RDB that is a set of assertions. In the evaluational one, a special type of inference mechanism, such as connection graph, etc., generates a sequence of goals, called plans, which can be evaluated by RDBMS. Though this one fits the current implementation of RDBMS, we ought to implement such an inference mechanism. Therefore, we propose a simple way in which PROLOG logic programming can interface with RDBMS. We give a *replace* program which is used to expand a condition in a goal statement so as to generate a sequence of evaluable predicates all of which must be retrieved in RDBMS. The program to amalgamate meta-language and object-language is given by using meta-level statements directly written in PROLOG itself.

Acknowledgement

The authors would like to thank Mr. K. Fuchi, the Director, Dr. K. Furukawa, the Chief, and Dr. S. Uchida, the Assistant Chief, of ICOT, for their useful comments and discussions on this research.

References

[1] ICOT (ed.): Outline of Research and Development Plans for Fifth Generation Computer Systems, May 1982.

[2] JIPDEC (ed.): Proceedings of International Conference of Fifth Generation Computer Systems, Oct. 19~22, 1981.

[3] Gallaire, H. and Minker, J.(eds.): Logic and Data Bases, Plenum Press, 1978.

[4] Feigenbaum, E. A.: The Art of Artificial Intelligence: Themes and case studies of knowledge engineering, IJCAI 1977, 1014~1029.

[5] Chang, C. L.: DEDUCE 2: Further Investigations of Deduction in Relational Data Bases in [3], 201~236.

[6] Sickel, S.: A Search Technique for Clause Interconnectivity Graphs, IEEE Trans. on Computers, Vol. C-25, No. 8. Aug. 1976, 823~835.

[7] Pereira, L. M., Pereira, F. C. N., and Warren, D. H. D.: User's Guide to DEC System -10 PROLOG, Laboratorio Nacional de Engenharia Civil, Lisbon, Portugal, Sept. 1978.

[8] Codd, E. F.: Relational Completeness of Data Base Sublanguages, in Courant Computer Science Symposium 6 Data Base System, Prentice Hall, 1972, PP. 65~98.

[9] Byrd, L., Pereira, F., and Warren, D.: A Guide to Version 3 of DEC-10 Prolog and Prolog Debugging Facilities, DAI Occasional Paper 19, Dept. of A.I., Univ. of Edinburgh, Scottland, 1980.

[10] Aho,A.V. and Ullman, J. D.: Universality of Data Retrieval Languages, ACM/SIGPLAN Conf. on Principles of Programming Languages, San Antonio, Jan. 1979, 110~117.

[11] Warren, D. H. D.: An Improved Prolog Implementations which Optimises Tail Recursion, DAI Res. Rep. No. 141, 1980.

[12] Bowen, K. A., and Kowalski, R. A.: Amalgamating Language and Metalanguage in Logic Programming, School of Computer and Information Science, Syracuse University, June 1981.

[13] Chakravarthy, U. S., Minker, J., and Tran D.: Interfacing Predicate Logic Languages and Relational Databases, Proc. of the 1st Int. Logic Programming Conf., Faculté des Sciences de Luminy Marseille, France, Sept., 14~17th, 1982, 91~98.

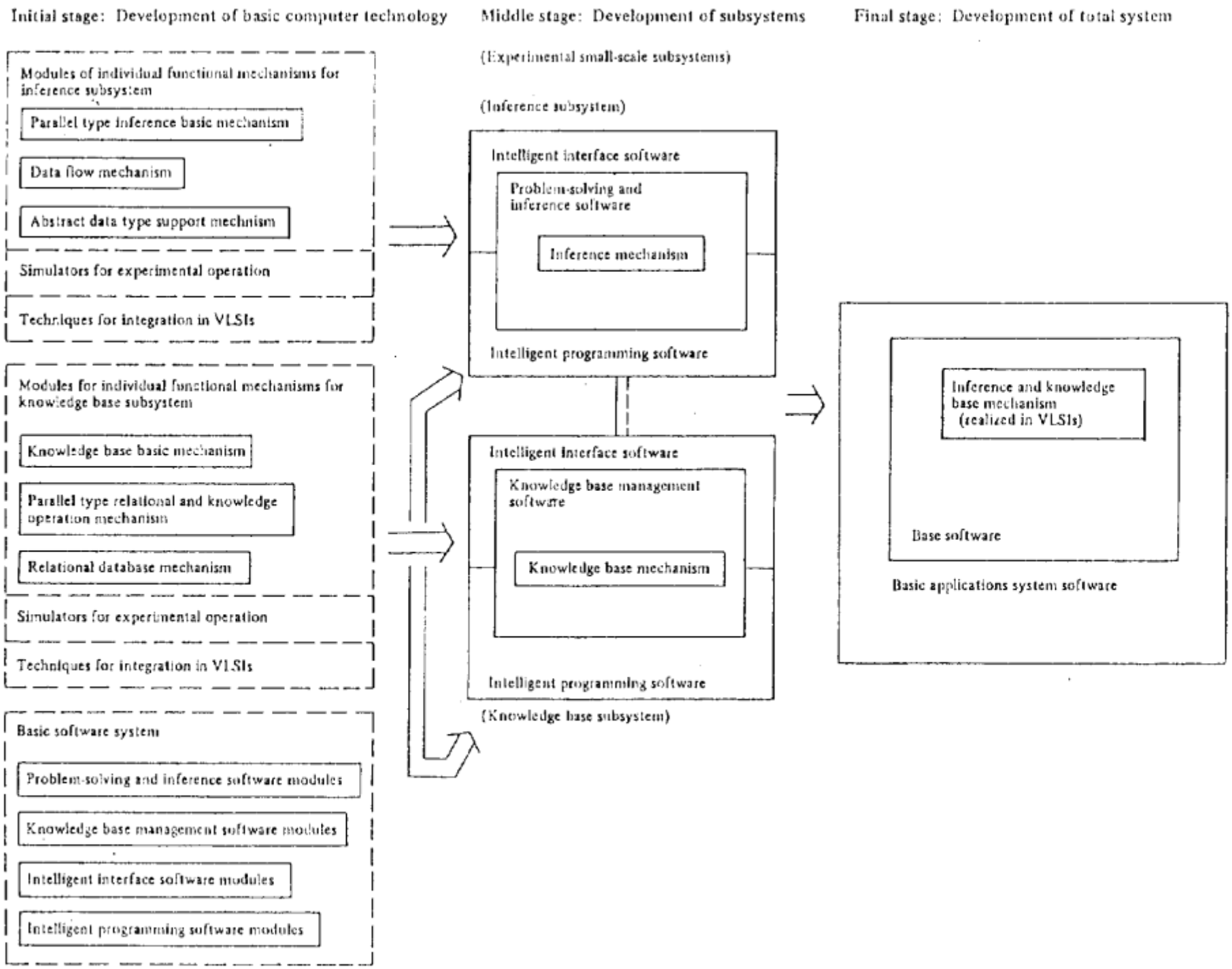

Fig. 1 Stages of fifth generation computer research and development

APPENDIX (A)

The meta-logical predicate *setof* (X, {Y^}Q, S) is an extension of PROLOG for generating multiple solutions, and means "S is the set of all instances of X {such that there exists a Y} such that the goal statement Q is provable, where the set is non-empty." The following program is implemented in the RT-11 PROLOG developed for the LSI-11 machine.

```
?- op(44,xfx,'^').

setof(X,P,L) :- functor(P,'^',N), arg(1,P,Y), arg(2,P,P1),
        (var(X), !, setof([Y,X],P1,L1); setof([Y|X],P1,L1) ),
        accum2(L1,L).
setof(X,P,L) :- gensym(s,S), vars(X,P,V), !,
        (P, S1 =.. [S,X|V], assert(S1), fail;
         S1 =.. [S,_|V], S1, accum1(S,V,L) ).

vars(X,P,V) :- functor(P,F,N), srch(X,P,N,V), !.

srch(X,P,N,[V1|V2]) :- N > 0, arg(N,P,Y), check(X,Y,V1),
        M is N - 1, srch(X,P,M,V2).
srch(X,P,N,V) :- N > 0, M is N - 1, srch(X,P,M,V).
srch(_,_,0,[]).

check(X,Y,Y) :- var(Y), !,(var(X), !, X \== Y; not(vmemb(Y,X)) ).
check(X,Y,Y) :- atom(Y), !.
check(X,Y,V) :- vars(X,Y,V), !.

accum1(S,V,L1) :- S1 =.. [S,X|V], retract(S1), accum1(S,V,L2),
        vunion([X],L2,L1), !.
accum1(_,_,[]).

accum2([],[]).
accum2([[_|L1]|L2],L3) :- accum2(L2,L4), vunion(L1,L4,L3).

vmemb(X,[Y|_]) :- X == Y.
vmemb(X,[_|L]) :- vmemb(X,L).

vunion([],S,S).
vunion([T|R],S,A) :- vmemb(T,S), !, vunion(R,S,A).
vunion([T|R],S,[T|A]) :- vunion(R,S,A).
```

APPENDIX (B)

The following questions and answers have been executed in the DEC-10 PROLOG V3.3 on DEC 2040.

A. Set operaitons

```
p(1,a).      q(1,a).
p(2,b).      q(2,b).
p(3,c).      q(3,e).
p(4,d).      q(4,b).
```

(given set P) (given set Q)

A1 Intersection — S = P ∩ Q

```
| ?- setof((X,Y),(p(X,Y),q(X,Y)),S).

S = [(1,a),(2,b)],
X = _29,
Y = _45 ;

no
```

A2 Union — S = P ∪ Q

```
| ?- setof((X,Y),(p(X,Y);q(X,Y)),S).

S = [(1,a),(2,b),(3,e),(3,c),(4,b),(4,d)],
X = _29,
Y = _45 ;

no
```

A3 Difference — S = P - Q

```
| ?- setof((X,Y),(p(X,Y),not(q(X,Y))),S).*

S = [(3,c),(4,d)],
X = _29,
Y = _45 ;

no
```

A4 Difference — S = Q - P

```
| ?- setof((X,Y),(q(X,Y),not(p(X,Y))),S).

S = [(3,e),(4,b)],
X = _29,
Y = _45 ;

no
```

A5 Cartesian product — S = P ⊗ Q

```
| ?- setof((X,Y,U,V),(p(X,Y),q(U,V)),S).

S = [(1,a,1,a),(1,a,2,b),(1,a,3,e),(1,a,4,b),
     (2,b,1,a),(2,b,2,b),(2,b,3,e),(2,b,4,b),
     (3,c,1,a),(3,c,2,b),(3,c,3,e),(3,c,4,b),
     (4,d,1,a),(4,d,2,b),(4,d,3,e),(4,d,4,b)],
U = _66,
V = _87,
X = _29,
Y = _45 ;

no
```

---

*
```
not(P):-P,!,fail.
not(_).
```

B. Relational algebra operations

B1 Projection

R(D_1	D_2	D_3)
a	2	f
b	1	g
c	3	f
d	3	g
e	2	f

```
r1(a,2,f).
r1(b,1,g).
r1(c,3,f).
r1(d,3,g).
r1(e,2,f).
```

```
| ?- setof(C,(A,B)^r1(A,B,C),S).

S = [f,g],
A = _45,
B = _66,
C = _24 ;

no
```

R[3](D_1)

f
g

B2 θ-restriction

R(A	B	C)
p	2	1
q	2	3
q	5	4
r	3	3

```
r4(p,2,1).
r4(q,2,3).
r4(q,5,4).
r4(r,3,3).
```

```
| ?- setof((A,B,C),(r4(A,B,C),B>C),S).

S = [(p,2,1),(q,5,4)],
A = _29,
B = _45,
C = _66 ;

no
```

R[B > C](A B C)

p	2	1
q	5	4

B3 θ-join

R(A	B	C)
a	1	1
a	2	1
b	1	2
c	2	5
c	3	3

S(D	E)
2	u
3	v
4	u

```
r2(a,1,1).      s2(2,u).
r2(a,2,1).      s2(3,v).
r2(b,1,2).      s2(4,u).
r2(c,2,5).
r2(c,3,3).
```

```
| ?- setof((A,B,C,D,E),(r2(A,B,C),s2(D,E),C=D),S).

S = [(b,1,2,2,u),(c,3,3,3,v)],
A = _29,
B = _45,
C = _66,
D = _87,
E = _103 ;

no
```

R[C = D]S(A B C D E)

	b	1	2	2	u
	c	3	3	3	v

B4 Division

R(A	B	C)
1	11	x
2	11	y
3	11	z
4	12	x

S(D	F)
x	1
x	2
y	1

```
r3(1,11,x).      s3(x,1).
r3(2,11,y).      s3(x,2).
r3(3,11,z).      s3(y,1).
r3(4,12,x).
```

```
| ?- setof(B,(setof(C,A^r3(A,B,C),S1),setof(C,E^s3(C,E),S2),subset(S2,S1)),S).

S = [11],
A = _73,
B = _24,
C = _52,
S1 = [x,y,z],
S2 = [x,y],
E = _181 ;

no
```

R[B,C][C = D]S = {11}.

---

* member(E,[E|R]).
member(E1,[E2|R]):-member(E1,R).
subset([],S).
subset([E|R],S):-member(E,S),subset(R,S).

APPENDIX (C)

The following amalgamation program '*replace*' is implemented in the RT-11 PROLOG.

```
 1
 2 /* Replace prolog prog. for EDB */
 3
 4 replace(_,[],[]) :- !.
 5 replace(_,[Head|Body],[Head|Body]) :-
 6         Head =.. [edb|Rest], !.
 7 replace(Prog,[Goal|Rest],AllReplace) :-
 8         prog(Clause,Prog),
 9         rename_vars(Clause,[Goal|Rest],[Head|Body]),
10         differ(Goal,Head,Diff),
11         add(Prog,Body,Rest,InterGoals),
12         apply(InterGoals,Diff,NewGoals),
13         replace(Prog,NewGoals,AllReplace).
14
15
16 /* Renamevars */
17
18 rename_vars([not(X)|L],_,[not(X)|L]) :- !.
19 rename_vars(Clause,Goals,Clause2) :-
20         search_var(Clause,Var1),
21         search_var(Goals,Var2),
22         Var1 = Var2,
23         gensym(Var1,Var3),
24         change_var(Clause,Var1,Var3,Clause1),
25         rename_vars(Clause1,Goals,Clause2), !.
26 rename_vars(Clause,_,Clause).
27
28 search_var([not(X)|L],Var) :-
29         !, search_var([X],Var).
30 search_var([X|L],Var) :-
31         X =.. [Pred|Args],
32         is_var(Args,Var).
33 search_var([X|L],Var) :-
34         search_var(L,Var).
35
36 is_var([X|Y],X) :-
37         name(X,[NLH|NLR]),
38         NLH > 'w'.
39 is_var([X|Y],Var) :-
40         is_var(Y,Var).
41
```

```
42 change_var([],_,_,[]).
43 change_var([not(X)|L1],Var1,Var3,[not(Y)|L2]) :-
44         change_var([X],Var1,Var3,[Y]),
45         change_var(L1,Var1,Var3,L2), !.
46 change_var([A|L1],Var1,Var3,[B|L2]) :-
47         A =.. [Pred|X],
48         changing(X,Var1,Var3,Y),
49         B =.. [Pred|Y],
50         change_var(L1,Var1,Var3,L2), !.
51
52 changing([],_,_,[]).
53 changing([Var1|L1],Var1,Var3,[Var3|L2]) :-
54         changing(L1,Var1,Var3,L2).
55 changing([X|L1],Var1,Var3,[X|L2]) :-
56         changing(L1,Var1,Var3,L2).
57
58
59 /* Difference */
60
61 differ(not(X),not(X),d([],[])) :- !.
62 differ(X,Y,d(Diff1,Diff2)) :-
63         X =.. [Pred|Arg1], Y =.. [Pred|Arg2],
64         diff(Arg1,Arg2,Diff1,Diff2), !.
65
66 diff([],[],[],[]).
67 diff([X|L1],[X|L2],Diff1,Diff2) :-
68         diff(L1,L2,Diff1,Diff2).
69 diff([X|L1],[Y|L2],[X|Diff1],[Y|Diff2]) :-
70         is_var([Y],Y),
71         diff(L1,L2,Diff1,Diff2).
72 diff([X|L1],[Y|L2],[Y|Diff1],[X|Diff2]) :-
73         is_var([X],X),
74         diff(L1,L2,Diff1,Diff2).
75
76 /* Add */
77
78 add(F,[NotTerm],Rest,InterGoals) :-
79         NotTerm =.. [not|SubGoals],
80         replace(F,SubGoals,NewSubGoals),
81         NewNotTerm =.. [not|NewSubGoals],
82         append(Rest,[NewNotTerm],InterGoals), !.
83 add(_,[Body],Rest,InterGoals) :-
84         Body =.. [edb|_],
85         append(Rest,[Body],InterGoals), !.
86 add(_,Body,Rest,InterGoals) :-
87         append(Body,Rest,InterGoals).
88
89 /* Apply */
90
91 apply(InterGoals,d([],[]),InterGoals) :- !.
92 apply(InterGoals,d([Var1|Rest1],[Var2|Rest2]),NewGoals2) :-
93         change_var(InterGoals,Var2,Var1,NewGoals1),
94         apply(NewGoals1,d(Rest1,Rest2),NewGoals2), !.
95
```