Database Management, Knowledge Base Management and Expert System Development in PROLOG

Kamran Parsaye

*Computer Science Research International**

ABSTRACT: The programming language PROLOG suggests a natural way of combining programming and deductive database queries by treating both programs and data as assertions in a database. We explore some issues in the implementation of databases and expert systems in PROLOG. We show that some simple extensions to PROLOG will allow for the convergence of many concepts from relational databases and expert systems into a uniform formalism for the management of both data and knowledge.

1. Introduction

The programming language PROLOG**, has been an interesting step in modern language design. By its nature of design, PROLOG *includes* a database and is hence a suitable language for database applications, particularly relational databases. Due to its symbolic nature and deductive capabilities, PROLOG is also a suitable language for expert systems implementations. Thus PROLOG seems a good candidate language for implementing both databases and expert systems.

In this paper we explore some issues which arise in the implementation of databases and expert systems in PROLOG. We show that some simple extensions to PROLOG will allow for the convergence of many concepts from relational databases and expert systems into a single formalism. This formalism can be used to approach both *database management* and *knowledge-base management* in a uniform manner.

Our extensions to PROLOG are, however, intended to preserve the flavor of PROLOG as a language. For instance, we show that the concept of functional dependency in relational databases is essentially equivalent to some PROLOG "cuts", that integrity constraints may simply be treated as PROLOG assertions, and that explanations and transparent reasoning in expert systems can be viewed as PROLOG execution traces. Many of the issues presented here grew out of the work on EDD (Expert Database Designer), a PROLOG based expert system for database design [Parsaye 82].

This paper is organized as follows: In section 1.1 we give a brief description of the language PROLOG. In section 2 we relate standard relational database concepts and terminology with PROLOG. We suggest an extension to PROLOG mode declarations, show the relationship between cuts and functional dependencies, and show how integrity constraints can be treated. Section 3 is devoted to PROLOG optimization issues for large database applications. We present a classification scheme for PROLOG clauses and propose the "independence assumption" for optimization. We also suggest how the notions of transactions

*) Author's Address: Computer Science Research International, 6420 Wilshire Blvd., Suite 2000, Los Angeles, CA 90048.

**) In this paper PROLOG essentially refers to the language originally defined by [Colmerauer 75] and implemented by [Warren 77].

and serializability can be easily introduced into PROLOG. In section 4 we focus on expert system applications. We propose a uniform view of database and knowledge-base management and illustrate two closely related approaches to knowledge representation in PROLOG. We also show how features such as explanations and transparent reasoning can be naturally programmed in PROLOG.

1.1 The Language PROLOG

In the context of this paper, it is particularly interesting to compare the development of PROLOG as a language to similar developments in data models and database languages.

Early database systems, e.g. IMS or CODASYL, use data structures such as trees or networks to store data. Users of IMS store and retrieve data by explicit insertion and retrieval operations which act upon tree structures, and in this sense deal with a *structure oriented language*. On the other hand more recent database systems, e.g. relational databases, hide the underlying data structures and implementation details from the user, and present associations and relationships in a non-navigational form.

Similarly, in programming languages such as FORTRAN, LISP or ADA one has to create data structures such as arrays, lists or stacks, store his data within these structures and later retrieve the data by navigational searches. On the other hand, in a *database oriented language*, such as PROLOG, the user can be unaware of the underlying implementation methods used for storing much of his data, and simply ask for data items to be stored and retrieved, just as he would ask a relational database system for storage and retrieval of data.

Software development in PROLOG can thus be mostly based on *"programming by assertion and query"* [Robinson 80], rather than by insertion and searchs of data structures. Moreover, the style of PROLOG programming is *declarative*, in the sense that a predicate (procedure) definition explicitly includes both the input and output parameters. Thus in PROLOG the distinction between input parameters and output parameters is much less prominent than in other languages, as seen by the examples below.

We now present a very brief and informal description of PROLOG, proceeding mostly by example. A detailed and comprehensive description of the language can be found in [Clocksin & Mellish 81], or [Pereira, Pereira & Warren 79].

The basic building blocks of PROLOG programs are *clauses*. A clause in PROLOG is a predicate name, called a *functor*, with some *arguments*. For instance

```
father(john, mary).
square(3, 9).
```

are clauses, where 'father' and 'square' are functors and 'john', 'mary', 3 and 9 are arguments.

Arguments may be constants or variables, and conventionally, non-numeric constants are denoted by lower case letters, while variables must start with uppercase letters, e.g. as in father(X, mary).

In PROLOG clauses can be *asserted* to be true, in which case they are included in the PROLOG *"database"*. The PROLOG database contains all facts which are asserted to be true. For instance,

assert(father(john, mary)).

will include father(john, mary) in the database and

retract(father(john, mary)).

will remove it.

In PROLOG, clauses are used to make *sentences*. A sentence in PROLOG may be a simple *unit* clause, such as father(john, mary), or it may involve the *conditional construct* denoted by " :- ", and better understood as "if".

For example, the conditional sentence

parent(X, Y) :- mother(X,Y) .

means that "for all X and Y", parent(X, Y) is true if mother(X, Y) is true. Thus essentially " A :- B " means that A is logically implied by B.*

Clauses on the right hand side of a " :- " can be joined together by "and" and "or" constructs denoted by "," and ";" respectively, as in

parent(X, Y) :- mother(X,Y) ; father(X,Y) .

which means that "for all X and Y", parent(X, Y) is true if either mother(X, Y) is true "or" father(X, Y) is true.

Let us make two simple technical notes here. First that sentences in PROLOG must end with a period. Second that due to the universal quantification above, the range of each variable in PROLOG is essentially a sentence, i.e. two occurances of the same variable name within two sentences are totally unrelated. The PROLOG compiler will internally rename variables to avoid conflicts.

Unlike equational programming languages, such as OBJ [Goguen & Tardo 79] or HOPE [Burstall et al. 80], PROLOG allows variables on the right hand side of a conditional which do not appear on the left hand side. Such variables are intended to be *existentially quantified*. For instance the sentence

grandfather(X, Y) :- father(X, Z), parent(Z, Y).

means that "for all X and Y", X is the grandfather of Y if "*there exists*" some Z in the database, such that X is the father of Z and Z is a parent of Y.

In PROLOG, conditional clauses may be stored in the database just as data are, i.e. programs are really treated as data in a database. This uniform view of both programs and data as items in a high level database is perhaps the major reason for the elegance of the PROLOG programming style.

Once one has adapted this database view of programming, one may naturally wonder about queries to the database. Simple queries may relate to simple facts such as: "Is father(john, mary) true in the database?", which may simply require a look up in the database, However, one may also ask more complex queries.

We generally refer to an attempt to answer a query in the PROLOG database as an attempt to *satisfy a goal* (or to *prove a goal*). For instance, in the example above "father(john, mary)" is the goal, and it can be satisfactorily proved if father(john, mary) has been asserted in the database.

One may also try to prove goals with variables, in which case PROLOG will try its best to find a match for the variables to satisfy the goal. For instance, an attempt to prove "father(X, mary)" will succeed provided that the condition (X = john) is true. Note that this is not an assignment (PROLOG is assignment free),

*) Logically speaking, PROLOG sentences are Horn Clauses [Horn 51].

but a binding of a variable to a value as in pure LISP. Such bindings are discarded upon the completion of the query.

Now, how about conditional clauses? Since the interpretation of the conditional construct " :- " is that the right hand side logically implies the left hand side, the validity of the left hand side can be established by proving the right hand side. This new goal may itself in turn be part of a conditional clause,..., and so on. Thus execution of programs in PROLOG essentially consists of attempts to establish the validity of goals, by chains of pattern matching on asserted clauses.

To prove a goal PROLOG searches its database for a clause that would *match* the goal, by using the process of *unification* [Robinson 65]. If a conditional clause whose left hand side matches the goal is found, PROLOG tries to satisfy the set of goals on the right hand side of ":-" in a left to right order. If no matching clause can be found, *failure* will be reported.

It must be noted that PROLOG includes no explicit negation symbol, and negation is essentially treated as *unprovability*, i.e. the failure to establish a goal from a set of axioms [Clark 79]. This closely resembles the closed world assumption [Reiter 78].

If PROLOG does not succeed in establishing a goal in a chain of deductive goals at a first try, it will *backtrack*, i.e. go to the last goal it had proved and try to satisfy it in a different way. For instance, suppose that we have the sentences (or program)

parent(X, Y) :- mother(X,Y) ; father(X,Y). (*)

grandfather(X, Y) :- parent(Z, Y), father(X, Z). (**)

and that the following facts have also been asserted:

father(john, mary).

father(paul, john).

mother(jennifer, mary).

Then to prove "grandfather(X, mary)", by using (*) and (**) above, first the goal "parent(Z, mary)" will be tried. This in turn will result in an attempt to prove "mother(Z, mary)" and will succeed with (Z = jennifer). Then, going back to the "grandfather" clause again, the next goal in the conjunction should be proved. So "father(jennifer, Y)" will be tried and will fail. At this point PROLOG will go back (i.e. backtrack), discard the assumption (Z = jennifer) and try to prove "parent(Z, Y)" again. This time (Z = john) will result, after trying "father(Z, mary)". Then the eventual binding (Y = paul) will be returned, after trying "father(X, john)".

Let us note that in the grandfather "program" here there are no explicit *input* or *output* parameters, i.e. one may either invoke grandfather(X, mary), or grandfather(paul, Y). This style of *declarative programming* in PROLOG can often be used to great advantage to develop software very rapidly. However, if a parameter in a program is always intended to be an input or output, the compiler can be signaled to generate optimized code by including *mode declarations* of the form

:-mode square-root(+, -).

which means that the square root function is never intended to be used to multiply a number by itself. Thus the user has the choice of running a program in both directions or not, as he sees fit.

On one hand, the series of steps taken by the PROLOG compiler in proving a goal essentially amount to *deduction*. On the other hand an attempt to prove a goal "father(X, Y)" can also be looked at as a procedure call to the predicate father. Thus the use of the term "logic programming" is quite apt here.

Calls in PROLOG can also be recursive, as in

connected(X, Y) :- edge(X, Z), connected(Z, Y).

which deals with connectivity in graphs described in terms of edges. The PROLOG compiler [Warren 77] uses tail recursion optimization to great advantage in such cases.

Now, for expression evaluation. In the author's opinion, one of the most inconvenient features of symbolic languages such as LISP has been the relation between quotation and evaluation. The PROLOG approach to evaluation is exactly the opposite of LISP, i.e. evaluation does not take place until it is forced to. This is specially relevant to arithmetic expressions and removes the need for quotes. Thus (2 + 3) can be evaluated to 5 when the need arises, by using the PROLOG infix operator "is", i.e. "X is (2 + 3)" binds X to 5. However, again note that this is not assignment.

Finally, one other feature of PROLOG which we need to mention is the "cut", denoted by "!". The cut is used to control backtracking in PROLOG. It is just treated as a goal itself, and can be used in any conjunction or disjunction of goals. Any attempt to satisfy "!" will succeed immediately for the first time, but will signal the compiler never to try it again. In fact an attempt to "retry" a cut will fail the parent goal invoking it, e.g. in

a(X) :- b(X,Y), !, c(Y,Z), d(Z).

backtracking can take place between c and d, but PROLOG will never backtrack to b. The cut can thus be used to gain efficiency and control in programs.

Many more examples of PROLOG programs, and a more detailed description of the language and its use may be found in in [Clocksin & Mellish 81], or [Pereira, Pereira & Warren 79].

2. PROLOG and Relational Databases

It is well known that relational databases can be viewed as logical predicates [Nicolas 77]. Essentially, each table in a relational database can be considered as the 'extensional' specification of a predicate. Each PROLOG predicate on the other hand, can be viewed as the 'intensional' specification of a relation or table. Moreover, it is also well known that most 'assertions', dependencies and integrity constraints in relational databases can be expressed as Horn Clauses [Fagin 80], which are essentially PROLOG sentences. Thus there is a natural correspondence between PROLOG and relational databases.

However, there are differences between existing relational concepts and PROLOG. In the next 3 sections we outline some of these differences and show, how with some simple extensions, they can be reconciled.

2.1 Schemas and Types

Relational databases usually rely on a typed system of logic and include schema information which determines the type and domain of attributes. PROLOG currently lacks these notions and relies on an untyped system of logic.

However, as [Nicolas 78] shows, an untyped system of logic can be easily

used to represent typed logic. For instance, the typed assertion

V XεINT p(X)

can be represented as the untyped sentence

V X (integer(X) & p(X)) ,

where & denotes conjunction.

The addition of schema and type information to PROLOG without affecting the flavor of the language is quite easy. PROLOG already includes mode declarations of the form:

:-mode employee(+, +, -).

which, for selected predicates, can be used to signal the compiler as to which parameters are intended as input and output.

To declare schemas, we suggest adding schema declarations of the form

:-schema employee(name, age, salary).

Similarly, we can add type information of the form

:-type employee(string[12], integer[3], integer[7]).

However, we believe that the inclusion of type information need not be mandatory and the user should be allowed to exclude type declarations for small relations, or when he sees fit.

The gain from having the declarations is two fold: on one hand they can be used for type checking and error detection, on the other they can be used by the compiler to achieve considerable enhancement in performance.

We feel that a major shortcoming of most current PROLOG implementations is that the compiler can not be informed that the argument to a square root function is intended to be an integer (rather than an arbitrary list), or that a social security number is a string of 9 digits. In most large database applications one needs to specify some type information and fixed length record sizes. We believe that before PROLOG can be used in a "real" large database application it should be extended to allow for the inclusion of type information within programs.

2.2 Functional Dependencies

PROLOG currently includes no notions of dependencies and normalization so far. These concepts were introduced into relational database theory since they are needed for design and for the avoidance of update anomalies. We believe that these concepts should be introduced into PROLOG in order to make it suitable for database applications. Moreover, in section 3.3 we show how functional dependencies can sometimes be used for optimization purposes.

Functional dependencies are simple enough to preserve the elegance of the PROLOG programming style. However, we feel that the addition of more complex dependencies, such as MVD's [Zaniolo 78] [Fagin 78] or EMVD's [Parker & Parsaye 80], may add an unnecessary amount of complexity to PROLOG programs.

Functional dependency information can be added to PROLOG in a manner similar to the type and schema information. However, interestingly enough, not only can this concept be incorporated into PROLOG quite naturally, but it gives rise to a different style of PROLOG programming.

In relational database terminology [Armstrong 77], the existence of a functional dependency A->B in a schema p(A,B) means that for each A there is only one B such that p(A,B) is true, e.g. X->Y in father(X, Y) means that each child

has at most one father.

We suggest the introduction of functional dependencies into PROLOG programs by declarations of the form

:- dependency(A->B) in p(A, B).

:- dependency(AB->C) in q(A,B,C).

At first a functional dependency may seem similar to a PROLOG construct of the form

..., p(A,B), !, ...

which fails the parent goal invoking p(A,B) if any goal following the cut fails. If a binding for A is supplied by the parent goal the cut is essentially equivalent to having the dependency (A->B) in p(A,B). In this case after failing p(A, B) once, one could not hope to find a new value for B by retrying p(A,B).*

However, if B is supplied by the parent goal and A is to be found by invoking p(A, B) then the the cut and the dependency are not equivalent, since the cut still forces the search to end. We feel that sometimes this use of cuts is against the general PROLOG philosophy that programs can be run in both directions when desired.

In general there has been a good deal of dissatisfaction with cuts in PROLOG anyway. We suggest that in many cases functional dependencies would be a much better alternative to cuts. Functional dependencies can often be used to write "cut-free", but efficient PROLOG programs, by directing the execution of programs in a manner which is dependent on the mode of procedure calls. Thus with the above functional dependency, in evaluating

q(A,B) :- ..., p(A, B), ...

there is an implicit cut after p(A, B) in the evaluation of q(a,B), but not in the evaluation of q(A,b). Moreover, note that the two sided declaration

:- dependency(A<->B) in p(A,B).

can be used to achieve a symmetric effect.

Of course, there are cases where one wishes to terminate the search after one unsuccessful attempt even though there is no dependency, in which case a cut will have to be used. However, this generally reduces the elegance and transparency of the "cut-free" PROLOG programming style.

2.3 Integrity Constraints

Enforcing database style integrity constraints expressed by Horn Clauses is very natural in PROLOG and is essentially a form of integrity enforcement by query modification [Stonebraker 75].

Clauses are usually added to the PROLOG database by the predicate 'assert', which adds almost anything to the database, without any integrity checks. To enforce integrity, we suggest the use of a predicate 'add' to assert facts which are subject to an integrity check. 'Add' is itself defined in PROLOG by

add(C) :- not(invalid(C)), assert(C).**

Conditions which should not be allowed in the database are indicated by the predicate 'invalid'. Thus, to enforce an integrity constraint on a predicate we add an assertion about invalidity. For instance, assume that we wish to enforce

*) Provided the integrity of the database has been preserved, as discussed in section 2.3.
**) Where 'not' denotes negation as unprovability.

the fact that an employee whose age is less than 19 can not earn over 100,000, i.e. that in

employee(Name, Age, Salary)

Salary should be less than 100,000 if Age is less than 19. We can simply add the assertion

invalid(employee(Name, Age, Salary)) :- (Age < 19) , (Salary > 100,000).

Thus the assertion

add(employee(johnson, 18, 120,000))

will fail, since

invalid(employee(johnson, 18, 120,000))

will succeed.

Functional dependencies are a special form of integrity constraint and will hence have to be enforced during addition of new data. A functional dependency (A->B) in p(A,B) can be enforced by simply adding the constraint

invalid(p(A, B)) :- p(X,B), not(eq(X,A)),

where 'eq' is defined by eq(X,X).

One may also wish to deal with the validity of responses, i.e. to ensure that returned values are consistent. Then one can define

return(A) :- A, not(invalid(A)).

to return results. Updates to the database can then be treated by combining additions and deletions.

The discussion above is aimed at integrity constraints that are usually placed on relational databases, i.e. constraints which essentially deal with unit clauses. We feel that enforcing constraints on non-unit clauses will often involve such a great deal of computation as to make it practically non-feasible.

3. Large PROLOG Databases

Having considered some high level database and language issues, we now focus on large database implementation and optimization issues relating to PROLOG.

Currently, all implementations of PROLOG either reside totally in core or rely on virtual memory. This proves to be sufficient for general programming and very small databases, but is certainly inadequate for serious database applications. However, we believe that with a suitable implementation strategy PROLOG can also be successfully used in conjunction with very large databases.

Moreover, since large databases are almost always shared by many users, we also need to consider PROLOG in a multiuser database context. We shall deal with these issues in the next three sections.

3.1 The Independence Assumption.

Much of the appeal of PROLOG has been the unification of the concepts of programming and querying into a single discipline by treating programs and data in a unified manner at the user level. However, while the user may be unaware of this distinction, we feel that for optimization purposes, a PROLOG implementor should separate these facts and deal with them accordingly.

Clauses in PROLOG can be classified into three categories:

a) *Non-unit Clauses*, i.e. clauses with both a left and a right hand side, e.g. clauses of the form p(A,B) :- q(A, C), r(C, B, X).

b) *Unit-Clauses with variables*, i.e. clauses with no right hand side, but with a variable argument, e.g. clauses of the form p(a,X).

c) *Ground-Unit Clauses*, i.e. clauses with no right hand side, and with no variable arguments, e.g. clauses of the form p(a, b, c).

Almost all of the information stored in current relational databases is of type c), while PROLOG 'programs' mostly contain clauses of types a) and b).

Currently most PROLOG implementations store and retrieve data by directly accessing a predicate's clause and (sometimes) hashing on one or more of the arguments. Moreover, almost all implementations use the same hashing method for clauses of class a), b) and c). In most large database applications this is simply an unacceptable implementation strategy since the size of and frequency of access and updates to data can be very different from the corresponding size and frequency for programs. Hence different hashing and indexing methods for these different categories of clauses are called for.

At first it may seem that the presence of a large number of 'database facts' of type c) and 'programs' of type a) for a given functor name can cause a problem since it may not be clear what form of hashing or indexing should be used for that functor name. However, we suggest that this need not be the case, and that the above classification can be used to implement large deductive databases more efficiently by making the following independence assumption:

For each given functor name, it is unlikely that there are a large number of Non-unit clauses and a large number of Ground-unit clauses at the same time. It is also unlikely that there are a large number of Unit-clauses with variables for any given functor name.

Assuming that Non-unit clauses are essentially 'programs' and Ground-Unit clauses are mostly 'data', the independence assumption means that programs and data are usually referred to with different functor names. The user may, if he wishes, indicate whether a functor name will be used for large database applications by a declaration of the form

:- largedata(employee(name, social-security-no, salary)).

Different hashing and indexing schemes may thus be used for these different classes. It would also be desirable to provide indices not only on the first argument but on other arguments of a predicate as specified by the user with a declaration of the form

:- index(B), index(C) in p(A, B, C).

which provide extra indices for B and C.

In this context, an interesting form of indexing for use in conjunction with deductive database systems has recently been proposed by [Lloyd 82].

3.2 Transactions, Concurrency.

Currently PROLOG is really only for single user personal databases, and includes no notions of transactions and concurrency control. Large databases are almost always accessed by more than one user, and there is a need for controlling the interleaving of the different user's programs in order to preserve the consistency of the database.

If PROLOG is to be used in large database applications, there will be need for sharing parts of databases between different PROLOG programs. This is not

directly related to expert system issues, but a PROLOG based expert system may need to access a shared database, say of patient medical records.

There will also be a need for including some form of transaction specification facility in PROLOG. There is also a need for the modification of most PROLOG implementations so that they would provide better interaction facilities with operating systems.

The introduction of transactions and concurrency control would require that some specified parts of a program be indicated as "atomic" actions, which are not interleaved with other programs. This is really a very simple point, and we are including it mostly for the sake of completeness.

To illustrate the concept of atomicity, consider a PROLOG transaction which performs transfers between accounts, i.e. the predicate

```
transfer(Account1, Account2, Amount) :-
        balance(Account1, X), balance(Account2, Y),
        Z is (X + Amount), W is (Y - Amount),
        retract(balance(Account1, X)), retract(balance(Account2, Y)).
        add(balance(Account1, Z)), add(balance(account2, W)).
```

The interleaving of the execution of this predicate with another user program such as

```
printsum(Account1, Account2) :-
        balance(Account1, X), balance(Account2, Y),
        W is (X + Y), print(W).
```

may result in inconsistent results. Thus the user needs to specify that he wishes 'transfer' to be an atomic action on the shared database.

We suggest adding simple declarations of the form

```
:- atomic(transfer(account, account, amount)).
```

to specify that a predicate should be implemented as an atomic transaction. The method of concurrency control can of course be left to the database operating system.

3.3 Implementing the 'Setof' Predicate

Some PROLOG implementations provide a predicate 'setof' which retrieves all instances of variables satisfying a predicate (or conjunction of predicates), e.g.

```
setof(X, (p(X, a, Y), q(Y,b), r(Y,c)), L)
```

retrieves into L all X for which p, q and r are true. Of course in many situations the order in which the predicates are evaluated can make a big difference. This form of conjunctive query optimization occurs quite frequently in database applications. Both System R [Astrahan et al. 76] and CHAT-80 [Warren & Pereira 81] deal with this issue by looking up relation sizes and reordering conjunctions.

We feel that the need for introducing this optimization into CHAT-80 is simply an indication of the fact that such a feature is missing from the basic PROLOG implementations. If PROLOG is to be used as a 'database' language, such feature would be necessary. It would not be hard to add such a feature essentially as CHAT-80 has implemented it.

Moreover, sometimes it might be possible to do even more optimization by using functional dependencies. A user can specify the order of the evaluation of conjuncts in his programs if he wishes, but any given order is not optimized for different modes of procedure calls. Again due to the PROLOG philosophy that

programs should be runnable in both directions it would be a good idea to allow the optimization to vary with the mode of the procedure call. Often it is possible to optimize in these situations if a functional dependency is known, e.g. if we know that

dependency(A->B) in q(A,B)

then in the evaluation of

setof(X, (p(X, Y), q(a, Y)), L)

it would usually be advantageous to evaluate q before p. This is also helpful in CHAT-80 like applications.

4. Some Expert System Issues

So far, we have discussed the appeal of PROLOG in database applications. Due to its symbolic nature and deductive capabilities, PROLOG is also a suitable vehicle for implementing expert systems. In the past few years, PROLOG has been the major language for expert system implementations in Europe. Some such systems, e.g. [Pereira & Porto 82], [Pereira et al. 82], [Darvas et al. 79], [Markusz 80] among others, offer encouraging results.

In the next sections we discuss the appeal of PROLOG's uniform approach to data and programs in expert system applications and and show how issues such as knowledge representation, explanations, transparent reasoning and inheritance can be dealt with.

4.1 Databases and Knowledge-bases

Currently, most expert systems dealing with databases have two distinct notions of *database management* and *knowledge-base management* [Davis & Lenat 82]. Often, the interaction between the knowledge-base and the database is not as smooth and well coordinated as one would wish.

As we have discussed before, PROLOG treats both programs and data in a uniform way. In expert systems applications, this can be looked upon as a single view of both 'data' and 'knowledge'. We suggest that this single view of both data and knowledge can be used to approach both database management and knowledge-base management in a uniform and elegant manner.

Looking back at the history of computing systems, one can view this as part of a general trend towards the development of very high level interfaces for interactive systems. The user interfaces of the computing systems of the 1960's were essentially based on the notion of *file management*, while since the early 1970's there has been a distinct trend towards high level *database management*. As [Ohsuga 82] points out, the user interfaces of the computing systems of the late 1980's and beyond are very likely to be mostly based on *knowledge-bases*. This signifies a general trend towards a uniform and high level style of interactive computing based on intelligent knowledge-based interfaces. We believe that PROLOG's uniform view of data and knowledge is a good basis for this gradual movement towards this form of knowledge-based interactive computing.

PROLOG is particularly useful in expert system applications which need to use large databases in one of the following ways:

a) They need to interact with large amounts of 'data' stored in databases, e.g. as in the RX system [Blum 82] which bases its inferences on a large database of medical case histories.

b) They need to use a database to store a large amount of 'knowledge' in

terms of a large number of rules which pertain to an area of expertise, e.g. expert systems which deal with a manufacturing environment [FGCS 81].

Of course, there are also many cases where both of the above conditions are satisfied. The advantage of using PROLOG in such applications is that the unified manner in which PROLOG approaches both data and programs (and in this case 'knowledge') results in a uniformity of design which facilitates the interaction between the human expert, the knowledge engineer and the expert system. As [Buchanan 79] points out, uniformity in design and representation is of great value in the development of expert systems.

We feel that in due course of time, most computing environments will be eventually liberated from the concept of a file system and will exclusively deal with unified databases and knowledge-bases. We also believe that due to its uniformity of approach, PROLOG is an excellent vehicle for this transition.

4.2 Knowledge Representation

Since PROLOG programs are essentially a subset of the sentences of first order logic, a natural knowledge representation method in PROLOG is a "logic flavored" knowledge representation method similar to MRS [Genesereth 81b]. Such representation has many advantages, but as we shall discuss later, it need not necessarily be the sole conceptual representation method for expert systems developed in PROLOG.

In the logical approach, the world is viewed in terms of 'predicates', and knowledge is essentially captured in terms of logical implications, i.e. production system like rules, or 'if then else' conditions. Such representation is in a way similar to the methods used by R1 [McDermott 80]. PROLOG sentences offer a convenient way of representing such rules, both in terms of 'deep' and 'surface' rules [Hart 82].

For instance, this form of representation is quite useful in the development of expert systems for diagnostic applications [King 82]. SUBTLE [Genesereth et al. 81a] uses an essentially similar approach. For example, a basic and general rule about the malfunctioning of structured components, say in an instrument diagnosis expert system, would be

```
malfunction(X) :- subcomponent(X, Y), malfunction(Y).
```

where the subcomponent information can itself be included in the database, as shown for example by

```
subcomponent(instrument, sensor).
subcomponent(instrument, connector).
subcomponent(instrument, display).
```

Specific structure relating to connectivity can be represented by assertions of the form

```
connector-input(X) :- sensor-output(X).
display-input(X) :- connector-output(X).
```

On the other hand, assertions of the form

```
malfunction(connector) :-
    connector-input(X), connector-output(Y), not(eq(X, Y)).
```

can be used to reflect the input/output relationships for the components.

In this example, note how easy it is to deal with the knowledge-base about the structure of the components just as one deals with a relational database containing parts and components information. Moreover, sometimes in the

course of diagnosis and repair of an instrument, the expert system may wish to gather information about the availability of "field replaceable units" from a common shared database. This can again be handled quite naturally by using the framework suggested in the previous sections.

However, the logical knowledge representation method need not be the only knowledge representation method used in conjunction with PROLOG. We feel that the "None for all, but any for some" truism of programming languages also applies to knowledge representation methods, i.e. that there is *no* knowledge representation method that is good for *all* applications, but that *any* knowledge representation method is perhaps good for *some* application. This suggests that one may use PROLOG in conjunction with different knowledge representation methods in different applications. We must, however, point out that the differences in these approaches are essentially conceptual and in many cases one approach may easily be translated into the other without much difficulty.

Another approach to knowledge representation would be a semantic network like approach, e.g. as suggested in [Brachman 80]. However, as [Deliyanni & Kowalski 79] point out, PROLOG's logical form can be closely linked to semantic network based knowledge representation techniques [Findler 80]. Moreover, in database applications, semantic network like representations may also be viewed as using some form of Entities and Relationships. EDD [Parsaye 82] uniformly uses the Entity-Relationship model [Chen 76] both for database schema design and for capturing the knowledge used in the design process by viewing Entity-Relationship diagrams as semantic networks.*

An example of a situation in which an Entity-Relationship like representation is intuitively appealing is in expert systems for office automation or in database design. As [Deliyanni & Kowalski 79] showed, in such cases one can simply capture the *schema* structure of the Entity-Relationship diagram by assertions of the form

```
relationship(employment, department, employee).
attribute(employee, name).
attribute(employee, social-security-number).
attribute(employee, department-number).
```

which reflect the fact that "employment", is a relationship between the entities "department" and "employee", and that "name", "social-security-number" and "department-number" are attributes of the entity employee. The translation of this representation to a logical form is very similar to the translation of Entity-Relationship diagrams into relational schemas, i.e. it involves the transformation of entities into relation names and attributes into arguments. For instance, the entity "employee" will be transformed into a relation schema

```
:-schema employee(name, social-security-number, department-number).
```

which can later be used to store information such as

```
employee(jones, 558 53 8973, departmet-4).
```

Thus, as is the case with Entity-Relationship diagrams and relational schemas, the logical and network-like representation methods can easily be translated into each other.

---

*) The fact that with very simple modifications, semantic networks diagrams can be easily transformed into Entity-Relationship diagrams has been part of computer science folklore for some time now.

Another issue that is sometimes quite important in knowledge representation is that of *subtypes* and *inheritance*. For instance, it is sometimes very useful to a user to deal with both "employees" and "managers", and record the fact that each manager is also an employee. There is a lot that can be said about such *polymorphic type structures* in theoretical terms [Parsaye 81], [Mac Queen 82], but in most practical cases these issues are quite simple to deal with.

As mentioned in section 2.1, types can be captured in untyped logic by the use of conjunctions, and thus such properties can easily be included in PROLOG programs by assertions of the form

employee(X) :- manager(X).

which specifies that each manager is an employee.

The inclusion of such conjunctions in PROLOG programs is no more easy or difficult than explicit type declarations for variables in a typed language, but this approach provides the flexibility of having or not having the types as desired.

4.3 Explaining Facts and Deductions

It is well known that relational databases can be viewed as logical theories [Nicolas 77], [Jacobs 81]. With this view, almost all data stored in, and queries posed to, current relational database systems deal with facts which are *ground literal* logical assertions or Ground-Unit PROLOG clauses. Such sentences correspond to what might be termed *who and what* facts and queries, e.g. "Who is the manager of department X" or "What is the salary of the oldest employee".

In expert system applications, 'knowledge' is captured in terms of facts which pertain to some form of expertise and need to be represented as non-ground literal clauses, i.e. Non-Unit clause sentences in PROLOG. Queries corresponding to such facts might be termed *how and why* questions, e.g. "Why did you recommend antibiotics for this patient", or "How did you know that this patient has diabetese".

Such queries are important since in the development of expert systems, it is often necessary to query the system about the knowledge used, and the series of deductive steps taken, in a deduction. This form of *transparent reasoning*, i.e. the ability of the expert system to explain and justify its actions and derivations is of utmost importance in the development of expert systems; without it the gradual enrichment of a simple set of rules into a non-trivial knowledge base would be almost impossible.

In such cases, it is not only necessary to explain the method of deduction and the knowledge used in the derivation of the answer, but to record *why* some piece of knowledge is in the database. For instance, it is usually necessary to record the actual patient case history which results in the addition of a rule to a MYCIN like system in order to facilitate future debugging [Shortliffe 76].

Once again, by invoking the uniformity of PROLOG's approach to knowledge and data, we suggest that explanations pertaining to both data and knowledge may be treated in a uniform manner. The basic idea is rather simple: each derivation in PROLOG essentially has the form of a proof tree whose leaves correspond to 'basic facts' or data, while the rest of the nodes reflect the structure of the proof.

The basic facts (i.e. the leaves) are obtained by some empirical means, e.g. laboratory tests, physicians observations, etc. The justification for these facts can be stored in terms of assertions in the PROLOG database itself, by using

assertions like justification(fact, reason), which record a basic reason for a basic fact, e.g.

:- justification(blood-count(johnson, 130), "test on 11/7/82").

Then the predicate "justify" can be used to justify basic facts by

justify(X) :- justification(X, Y), print(Y).

Such method may also be used for justifying the addition of non-unit clauses to the PROLOG database, e.g.

:- justification(rule 133, "patient case history 173").

Moreover, the steps involved in the deduction are essentially those steps involved in pattern matching and unification. Most PROLOG implementations offer debugging facilities which allow the user to trace the steps in the execution corresponding to a certain predicate. We suggest that similar technique can also be used in explanations, e.g. suppose we have

```
grandfather(X, Y) :- father(X, Z), parent(Z, Y).
parent(X, Y) :- mother(X,Y) ; father(X,Y).
father(john, mary).
mother(mary, paul).
```

A first level explanation of "grandfather(john, paul)" can be obtained by following the steps of the unification, i.e.

grandfather(john, paul) *since* father(john, mary), parent(mary, paul).

A further level of explanation may then be obtained by

parent(mary, paul) *since* mother(mary, paul).

Now let "trace(X, Y)" give Y as the top level goals which were used in the derivation of X. The predicate "justify" can then be extended to the trace and be used to give explanations by using "explain", where

```
explain(X) :- justify(X).
explain(X) :- trace(X, Y), explain(Y).
explain(X ',' Y) :- explain(X), explain(Y).
```

Another interesting issue is to ask "*why not*" questions, e.g. "Why is not john the father of mary?". A simple answer to this can be that this fact is non-existent in the database, but sometimes there may be need for the display of partial deductions that fail. This is quite interesting to program in PROLOG, and is left to the reader as an exercise.

There are of course many other issues that need to be dealt with in the context of multi-level explanations. A number of these issues are discussed in [Swartout 81], and a good deal more work remains to be done on the subject.

5. Conclusions

We have shown how PROLOG can be used to arrive at a uniform and high level approach to both database management and knowledge-base management. We have also pointed out the appeal of this single approach to the management of both data and knowledge in expert system applications. As a language, PROLOG holds a lot of promise. We believe that with the advent of architectures more suited to its implementation [FGCS 81], PROLOG will become a dominant force in computing in the 1980's.

6. Acknowledgements

I wish to thank David Warren and Fernando Pereira of SRI, Stott Parker of UCLA, Edward Katz, Heinz Breu, Robert Fraley and Martin Liu of Hewlett Packard, Robert Blum and Gio Wiederhold of Stanford University, Jonathan King of Symantec and Antonio Porto of the University of Lisbon for helpful comments and discussions. Some of this work was performed while the author was at the Computer Research Center, Hewlett Packard Laboratories.

References

[Armstrong 74]

Dependency Structures of Database Relationships, Proceedings of the 1974 IFIP Congress, Amsterdam.

[Astrahan et al. 76]

System R : A Relational Approach to Data Management, ACM Transactions on Database Systems, Vol 1, No 2.

[Blum 82]

The RX Project, Computer Science Department, Stanford University.

[Brachman 80]

Knowledge Representation in KLONE, in [Findler 80]

[Buchanan 81]

Research on Expert Systems, Heuristic Programming Project Report, Stanford University.

[Burstall et al. 80]

HOPE, An Experimental Functional Programming Language, Proceedings of the 1980 LISP Conference, Stanford University.

[Chen 76]

The Entity Relationship Model, ACM Transactions on Database Systems, Vol 1, No 1.

[Clark 78]

Negation as Failure, in [Gallier & Minker 78].

[Clark 80]

PROLOG, A language for Implementing Expert Systems, in 'Machine Intelligence 10', edited by P. Hayes and D. Michie.

[Clocksin & Mellish 81]

Programming in PROLOG, Springer Verlarg.

[Colmerauer 75]

Les Grammaires de Metamorphose, Groupe d'Intelligence Artificialle, Marsille-Luminy, 1975, reprinted in Lecture Notes in Computer Science Vol. 63.

[Dahl 82]

On Database Systems Development through Logic, ACM Transactions on Database Systems, Vol 7. No 1.

[Deliyanni & Kowalski 79]

Logic and Semantic Networks, Communications of the ACM, Vol 22. No 3.

[Darvas, et al. 79]

A Logic Based Program for Predicting Drug Interactions, International Journal of Biomedical Computing, Vol 9.

[Davis & Lenat 82].

Knowledge-Based Systems in Artificial Intelligence, McGraw Hill, New York.

[Fagin 78]

A New Normal Form for Relational Schemas, Research Report, IBM San Jose Research Laboratory.

[Fagin 80]

Implicational Dependencies, Proceedings of the 1980 ACM Conference on Foundations of Computer Science.

[Forgy 81]

OPS5 User's Manual, Department of Computer Science, Carnegie-Mellon University.

[FGCS 81]

Proceedings of the Fifth Generation Computer Systems Conference, Tokyo, Japan.

[Findler 80]

Associative Networks, North Holland.

[Furukawa 81]

Problem Solving and Inference Mechanisms, in [FGCS 81].

[Gallier & Minker 78]

Logic and Databases, Plenum Press, New York, 1978.

[Genesereth et al. 81a]

SUBTLE Reference Manual, Computer Science Department, Stanford University.

[Genesereth et al. 81b]

MRS Reference Manual, Computer Science Department, Stanford University.

[Goguen & Tardo 79]

An Introduction to OBJ, Proceedings of IEEE Conference on Reliable Software, Boston.

[Hart 82]

The Future of Artificial Intelligence, Artificial Intelligence Technical Report, Fairchild Laboratories.

[Horn 51]

On Sentences which are True of Direct Unions of Algebras, Journal of Symbolic Logic, Vol. 16.

[King 82]

Knowledge-Based Diagnosis and Repair, Internal Memorandum, Hewlett Packard Laboratories.

[Kitagawa 82]

Japan's Annual Reviews in Computer Science and Technologies, North Holland.

[Kowalski 79]

Logic for Problem Solving, North Holland.

[Kowalski 81]

Logic as a Database Language, Department of Computing, Imperial College London.

[Jacobs 81]

Applications of Logic to Databases, Department of Computer Science, University of Maryland.

[Lloyd 81]

Implementing Clause Indexing in Deductive Database Systems, Technical Report, Department of Computer Science, University of Melbourne.

[Mac Queen 82]

A Polymorphic Type System, Technical Report, Bell Laboratories.

[Markusz 1980]

Applications of PROLOG in Many-storied Panel House Design, Proceedings of the 1980 Logic Programming Conference, Hungary.

[Mc Dermott 80]

R1 : A Rule-Based Configurer of Computer Systems, Technical Report, Computer Science Department, Carnegie-Mellon University.

[Michie 80]

Expert Systems in the Micro-electronic Age, Edinburgh University Press.

[Nicolas 78]

Logic and Databases, in [Gallier & Minker 78].

[Ohsuga 82]

Knowledge-Based Systems as a New Interactive Computer System of the Next Generation, in [Kitagawa 82].

[Parker & Parsaye 80]

Inferences Involving Embedded Multi-Valued Dependencies and Transitive Dependencies, Proceedings of the 1980 ACM SIGMOD Conference.

[Parsaye 81]

Higher Order Abstract Data Types, Ph.D. Dissertation, Computer Science Department, UCLA.

[Parsaye 82]

EDD, an Expert System for Database Design, Internal Report, Computer Research Center, Hewlett Packard Laboratories.

[Pereira 82]

Can Drawing be Liberated from the Von Neuman Style?, Technical Report, SRI International.

[Pereira, Pereira & Warren 77]

DEC-10 PROLOG users Guide, Department of Artificial Intelligence, University of Edinburgh. [Pereira et al. 82] Orbi

[Pereira & Porto 82]

A PROLOG Implementation of a Large System on a Small Machine, Proceedings of the 1982 Logic Programming Conference, Marseille.

[Reiter 78]

On Closed-world Databases, In [Gallier & Minker 78].

[Robinson 65]

A Machine-oriented Logic Based on the Resolution Principle, Journal of the ACM, Vol. 12, No. 1.

[Robinson 80]

Programming by Assertion and Query, in [Michie 80].

[Shortliffe 76]

Computer Based Medical Consultations: MYCIN, Elsevier, New York.

[Stonebraker 75]

Implementation of Integrity Constraints on Views by Query Modification, Proceedings of the 1975 ACM SIGMOD Conference.

[Suwa 81]

Knowledge Base Mechanisms, in [FGCS 81].

[Swartout 81]

Explaining and Justifying Expert Consultation Programs, Proceedings of the 7th IJCAI.

[Warren 77]

Implementing PROLOG - Compiling Predicate Logic Programs, Research Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh.

[Warren 81]

Efficient Processing of Interactive Relational Database Queries Expressed in Logic, Proceedings of the 1981 VLDB.

[Warren, Pereira & Pereira 77]

PROLOG, The Language and Its Implementation Compared with LISP, Proceedings of the ACM Symposium on AI and Programming Languages.

[Warren & Pereira 81]

The CHAT-80 System, Technical Report, Department of Computer Science, University of Edinburgh.

[Zaniolo 78]

Studies on Relational Databases, Ph.D. Dissertation, Department of Computer Science, UCLA.