KNOWLEDGE REPRESENTATION IN AN EFFICIENT DEDUCTIVE INFERENCE SYSTEM

E. P. Stabler, Jr.
E. W. Elcock

University of Western Ontario
London, Canada

ABSTRACT

Efficient response to queries addressed to a large data base is an important problem of knowledge representation. The problem and various solutions have been well researched for certain "conventional" (e.g. relational) data models. The analogous problem has been tackled and solutions similar in spirit to those for relational models developed for data bases and queries expressed in Horn clause systems such as Prolog with the severe constraint that the "data base" is a set of ground instances of assertions.

The situation becomes more interesting and challenging when the data base is deductive: e.g. a Prolog first-order theory. Basically the interest is in finding an automatic way of representing the first-order theory which facilitates the dynamic reordering of residual literals and the selection of the next goal to be evaluated based on a changing measure of the cost of evaluating each goal in the residual query.

The current paper presents partial solutions which can be used to obtain dramatic reductions in search times. The paper also identifies some remaining and difficult problems.

The methodology was designed with the processing of natural language queries in mind, but it is quite general in its domain of application.

KNOWLEDGE REPRESENTATION IN AN EFFICIENT DEDUCTIVE INFERENCE SYSTEM

E. P. Stabler, Jr.
E. W. Elcock

University of Western Ontario
London, Canada

There is an increasing demand for large database systems that provide efficient inference capabilities. These are obviously needed in question answering systems and expert systems, but their potential range of application is really very wide. It is an advantage to allow any database user a uniform view of both explicit and implicitly represented information. Accessing the database through a deductive inference system offers the possibility of such freedom, first, in its ability to decide whether any particular proposition follows from what is represented in the database, and second, in its ability to use rules, meaning postulates and definitions of new terms and relations as nonlogical axioms in making such decisions. Thus, rather than having to search for particular pieces of information, the user can simply ask whether or not a particular proposition follows from the database: general rules can be introduced to cover large classes of particular facts and to define new terms that might be used in queries.

Keeping the deductive access to a large database efficient requires, in the first place, that we deal with some of the standard database management problems, viz., the problems of making search efficient and of optimizing queries to minimize the need for search. These problems are paricularly pressing when the database and access system are to be embedded in a larger design, such as a question answering system. In this context the system must interface with natural language processors, rather than with the typical brilliant and insightful human database user who can learn how to avoid the system's weak spots. As a result, database queries cannot be expected to arrive in a form that is optimal from the point of view of efficiency. In this report we will show how such standard database management problems can be handled in PROLOG, one of the most efficient and most widely known theorem proving systems.

---

* The software described in this paper was designed and implemented by a research group which included the authors, D. Wyatt, and A. Young. This work is also described in Elcock et al.(forthcoming) and Stabler (1982).

PROLOG preliminaries. Since PROLOG is fairly well known and there are good introductions to the language (e.g., Clocksin and Mellish, 1981) we will only briefly review some important features. PROLOG is basically a Horn clause theorem prover. It also has metalogical facilities that can provide higher order effects, and of course nonstandard effects can be obtained by quantifying over possible worlds (cf. Moore, 1980). A clause is a first order prenex formula whose prefix consists of (only) universal quantifiers and whose matrix is a disjunction of literals, where a literal is an atomic formula or the negation of an atomic formula. Since the order of universal quantifiers makes no difference, they do not need to be written down. A Horn clause is a clause whose matrix contains at most one positive literal. The restriction to Horn clauses does not, in principle, prevent us from expressing anything that can be expressed in first order logic. The relevant results are these: for any formula F of first order predicate calculus, there is an easily constructible set S of clauses which is inconsistent if, and only if, F is (see e.g., Chang and Lee thm.4.1.); and for any set S of clauses there is an easily constructible set H of sets of Horn clauses such that there is an inconsistent set in H if, and only if, S is inconsistent (see e.g., Henschen and Wos, 1974). In spite of this generality in principle, though, some problems are much more feasible and natural when expressed in non-Horn clause form.

The restriction to the Horn clause subset of first order logic is not the only special logical problem that faces a PROLOG database system. In the first place, we should note that PROLOG does not immediately provide ideal inference capabilities even within the Horn clause logic: it's failings are the familiar ones. It is well known that "Horn sets" are not decidable (Hermes, 1965), and so of course PROLOG cannot decide whether an arbitrary query of its Horn clause logic follows from the database or not, even given unlimited time and space. And although it is easy to design proof methods for Horn clause logic which are complete in the sense that they will find a proof of an arbitrary sentence if there is one, the most efficient theorem provers, like PROLOG, are not complete. They are even unsound in the sense that they will sometimes claim to have found a proof when there is no valid proof. Let's consider these problems briefly before considering the more standard database problems.

Soundness. It is well known that PROLOG will sometimes produce an invalid proof. For example, given the database "p(X,X).", PROLOG will say that the query "p(Y,f(Y))." follows. The instance that follows, according to the PROLOG system, is the one in which Y=f(f(f...(f(Y))))).... (Since this is an infinite expression, there will be trouble if PROLOG tries to print it out.) But obviously,

"(Ey)(P(y,f(y)))" does not follow from "(x)(P(x,x))". PROLOG gets this incorrect result because it does not do the "occurs check" in the course of unification. What it does is this. When confronted with the query "p(Y,f(Y))" it tries to match it with the database clause "p(X,X)". It begins with the first argument; in effect, "Y" is substituted for "X". The result is "p(Y,Y).", and this is identical to the query up to the second argument. So now PROLOG tries to match the second arguments, which it does by substituting "f(Y)" for "Y". As a result, Y=f(f(f...(f(Y)))...., and the query is judged to be an instance of the database clause. Strictly speaking, this matching process is not the unification which is employed in sound resolution procedures, because a variable cannot properly be unified with any term which contains that variable. Implementing unification correctly would involve performing an "occurs check" to make sure that the variable does not occur in the term it is being unified with. Performing this check in every unification step is expensive, especially when the terms being unified are large. Since the matching process without an occurs check is so much cheaper, and since it is sufficient in most cases, most PROLOG implementations do not use strict unification. As noted above, assuming that we do not want to just tolerate errors, this means that the users of these systems must make sure they are not accepting conclusions based on unsound inferences. There are a number of ways to do this.

One way to avoid unsound inferences is simply to require that only ground clauses occur in the database. This restriction is perfectly straightforward, and it is obviously adequate since the database would then not contain any variables which might occur in terms they would be matched with. The problem is that this restriction is obviously going to eliminate the features which make logic programming languages particularly attractive. (Consider, for example, the features mentioned at the beginning of this paper.)

There are other similar and less restrictive strategies. We can hope that programmers experienced with PROLOG will learn how to avoid creating a database in which unsound inferences will be made. If an unsound inference is going to cause trouble, they should block it. The problem then is not with the "system" database which is provided by programmers, but with database clauses which might be provided by naive users. We do not want to require the users to understand and attend to such things as the peculiarities of the PROLOG matching process. So we can either provide a complete system to which the user cannot add information, or we can require that any information added be ground clauses. The "system" database would then be created by programmers familiar with PROLOG's matching process, and the "user" database, if there is one, would not add any dangers of unsoundness. This restriction on the users' database would certainly to be felt,

however; it severely constrains what the user can do with the system. He would not, for example, be able to add definitions of new relations in terms of relations already provided by the system. The only other alternative that seems to be available, though, would be to do the occurs check whenever non-ground clauses that could conceivably cause an error are used. Since the main thrust of the present project is to allow the database users the real advantages of accessing a database through an inference system (without errors, even very unlikely ones!), this last strategy is the only acceptable one, and is currently being explored.

Completeness. The second problem that we would like our system to deal with as well as possible is that of avoiding attempts to find proofs which are beyond the theoretical capabilities of PROLOG. We have already noted the obvious point that mere completeness is not going to do us any good if the proof procedure is just not feasible. But the point of interest is that if finding a proof of some result is beyond PROLOG's theoretical capabilities, it is of course also beyond its practical capabilities. It is a good strategy to try to keep the whole class of proofs that might be sought within the theoretical capabilities of PROLOG, and then to keep those proofs as efficient as possible. Sometimes a simple change in the database, query, or proof strategy that brings a result within the theoretical capabilities of the system also suffices to bring the result within the practical capabilities of the system.

The following familiar sort of example illustrates this situation. (This example is taken from Moore(forthcoming), where it is used to illustrate the related problem of forward vs. backward chaining.) One of the standard ways to define a relation is with "base rules" and "induction rules." For example, the one-place relation or property of being Jewish might be partially defined with a list of people who are Jewish and with the rule from the Talmud that a person is Jewish if the mother of that person is Jewish, as follows:

```
jewish(bar-hillel).
jewish(X):-jewish(mother(X)).
```

Given this database, PROLOG will properly indicate that there is a proof of the query "jewish(bar-hillel).". If, however, the clauses in the database are reversed, putting the "induction rule" before the "base rule," PROLOG will never succeed in finding a proof of this query. Because it uses a depth-first proof strategy and selects the first database clause first, it would never get to the second rule, the base rule, which it would need to use. It would "loop," using the first rule, the "induction rule," over and over again. Since this sort of situation is quite common, we can adopt the

strategy of always putting "base" rules before "induction" rules. The problem is to recognize them. A crude approximation that will handle this case is to check each clause that is going into the database to see if it is a simple assertion, a unit clause with an empty body. If it is, put it at the beginning of the list of clauses which have the same predicate; otherwise, put it at the end of the list. (This is one of the things which is done by our predicate "update" which will be described in more detail later.) It should be noted, however, that this ordering strategy will not work in cases where the "base" rules are not simple assertions, and it will not work in cases in which there is more than one "induction rule." There are cases of incompleteness which cannot be removed by any reordering of database clauses. (Cf. Elcock, 1982; 1983.) Thus, our implementation of this ordering strategy is not motivated so much by completeness considerations as by feasibility: it is generally cheaper to find solutions using unit clauses, so these should be considered first.

Feasibility. Problems which are at present effectively insurmountable also seem to face the general goal of staying within the practical limits of the system. The use of a language that has a formal, logical interpretation is no panacea for the standard sorts of programming problems; we do not have any mechanical method for transforming logically correct but inefficient code into correct and efficient code. The ordering method just described will help in some cases. Another thing that is done (by "update") to improve efficiency is that whenever a clause is added to the database, all instances of that clause are deleted. So, for example, the addition of "p(X)." to the database will cause "p(a)." to be deleted. And the addition of "p(X,Y)." will cause "p(X,X)." to be deleted. So a certain easy to find redundancy is automatically eliminated. Apart from such simple steps as these, though, there is not much that can be done cheaply and easily to enlarge the class of feasible proofs except to provide as much time and space as is practical, to minimize the need for unnecessarily long searches, and to make searches of the database as efficient as possible. Search efficiency can be improved by indexing the database; unnecessary search can be eliminated with appropriate goal selection strategies and intelligent backtracking. Each of these methods will now be considered in turn. Notice that none of them are theorem proving matters; they are metalogical operations that change the set of axioms from which we may draw inferences. They can be taken care of automatically, out of the sight of the user. The user should see only the improved efficiency.

Indexing the database. A standard technique for making search efficient involves indexing the units of information so

that when an item is needed the whole memory does not need to be searched; instead the location of the needed information can be looked up in an array or hash table. The DEC-10 PROLOG interpreter indexes database clauses according to their "head" predicates, i.e., according to the predicate in the head of each clause (Pereira et al., 1978). But when a relation is large, when there are many clauses with a particular head predicate, the searches will still be long. In this situation, the standard strategy is to start secondary indexing on the arguments of the relations. A database that is indexed for every argument of every relation is said to be totally indexed or totally inverted. Some PROLOG implementations, such as IC-PROLOG, provide facilities for indexing according to the principal functor of arguments to the head predicates in the database (Clark and McCabe, 1982). And in systems like interpreted DEC-10 PROLOG, secondary indexing effects can be obtained simply by building auxiliary predicates which incorporate names of the principal functors of the arguments. This technique was used in the Edinburgh Chat-80 system (Warren, 1981; Warren and Pereira, 1981), and we used it in our work.

Goal selection strategies. The order in which the goals of a query are solved can make a substantial difference in resource use. Suppose, for example, that the database has 4000 clauses with the predicate "g1" and 1 clause with the predicate "g2", and that all of these are ground clauses. Then, given the left-to-right selection method that is standard in PROLOG, and assuming that the database is totally indexed, it is much more efficient to evaluate the query,

```
g2(X,Y),g1(X,Y).
```

than it is to evaluate the query,

```
g1(X,Y),g2(X,Y).
```

Evaluating the latter query could involve an enormous amount of backtracking. Evaluating "g2(X,Y)" first, on the other hand, immediately provides the only instances of "X" and "Y" which could possibly satisfy the query. The indexing will allow this instance to be checked without a long search, and, in any case, backtracking is more expensive than a simple search for a matching head predicate. So in general, we want to evaluate the least expensive goals first. When the database is all ground clauses and the query has variables in all argument places, we can let the cost of a goal be the size of the relation, i.e., the number of clauses in the database whose heads have the same predicate as the goal. The cost function should be more elaborate, however, when the database contains clauses with variables (or terms containing variables) or the query contains goals with non-variables.

Let's consider first the elaboration of the cost function which is needed to allow for queries with instantiated arguments. If a predicate is indexed in the database, then

any "user" query of that predicate will be solved by first converting it into its indexed form and then finding a solution to that "indexed" query. In a totally indexed database the cost of solving the original query will not in general depend on the size of its main predicate, but rather on the size of the sets of arguments that occur in each of the n-positions of any n-place predicate in the query, since these are the arguments to the indexed predicates. In order to estimate the expense of finding a solution to a query (in a manner which will be described below) we can keep records of the sizes of the sets of arguments that occur in each place of every predicate. When all the database clauses are ground clauses, calculating the sizes of these sets is straightforward. The sets simply include all the different terms that occur in the relevant argument positions.

This brings us to the question of how to elaborate the cost function to make it appropriate for a totally indexed database that is not restricted to ground clauses. In this situation, not all of the possible instantiations of any particular argument position need be explicitly available; some of them will only be found by the inference process. We do not want to have to calculate all of the possible instantiations of each predicate, so we need some reasonable way of estimating the number of distinct terms that could occur in each argument position. The details of the calculation will not be described here, but roughly, we make worst-case assumptions that allow us to calculate the maximum number of possible distinct provable instantiations of each predicate. And then, thinking of each different predicate as a relation, we want some reasonable way of calculating the relation size. Again, we calculate relation sizes by making a worst-case estimation of the number of solutions one would be able to find to the query consisting of any particular predicates followed by the appropriate number of variables. We calculate these estimates and revise them when new information is added as part of the "updating" process. Given these estimates, we are able to use the same cost estimation formula as was used in the Chat-80 system for ground clause databases. The cost of solving a goal is defined as the size of the relation divided by the product of the argument domain sizes associated with argument positions that are instantiated at the time a solution is sought.

Notice that, given this definition, the cost of a goal may change when other goals in the query are solved. For example, in solving the query,

```
g1(X,Y),g2(Y,Z),g3(Z,a).
```

the solution of the first goal will instantiate the first argument of the second goal, making it cheaper to solve. And the solution to the second goal will leave no uninstantiated arguments in the third goal. So if we want to plan our queries in such a way that the cheapest goal will always be

the next one solved, we will have to anticipate the instantiation of the relevant variables. This process interacts with the backtracking strategies described below, so let's consider those before describing how this query planning should be done.

Selective backtracking. Sometimes PROLOG will do a lot of unnecessary backtracking in the course of finding the set of solutions to a query. Consider, for example, the query,

```
bagof(X,h(X),B).
```

where the unary predicate "h" is defined by the database clause,

```
h(X):-g1(X),g2(Y).
```

And suppose the database provides some number n of solutions to the first goal, "g1(X)", and some very large number m of solutions to "g2(Y)". In finding the list B of solutions to "h(X)", a solution to the first goal "g1(X)" will be found; then a solution to the second goal "g2(Y)" will be found and the instance of "X" will be put in list B. The system will then backtrack to find all m solutions to the second goal, putting the first solution to the first goal in the list B each time. Since we are only interested in getting the instances of "X" which satisfy the goals given, it is just a waste to get each such solution m times. We could use "setof" instead of "bagof" to get a nonredundant list of solutions, but this query also wastes the time to get all the redundant solutions before deleting them. Interchanging the positions of "g1(X)" and "g2(Y)" does not improve things. And simply putting a cut into the original query somewhere will also not achieve the goal of getting a complete set of the instances of "X" without this wasted effort. (In this case we could interchange the goals and put a cut between them, but this sort of solution will not always be available, as the examples below will illustrate.) Because it shares no variables with the head of the clause, the goal "g2(Y)" is, in effect, an independent subproblem; it must have a solution, but this is all we need to know to find all of the solutions to "h(X)".

Precisely the same situation arises if instead of having a definition of "h", we simply ask,

```
bagof(X,(g1(X),g2(Y)),B).
```

We would like to be able to avoid the unnecessary backtracking in all such cases.

This problem was handled in the Chat-80 system by putting independent subproblems inside braces, and then changing the PROLOG interpreter so that it would evaluate queries containing such braces appropriately. We used the standard interpreter and used new rules with cuts to achieve the same effect. Thus, instead of evaluating a query like

```
bagof(X,(g1(X),g2(Y)),B).
```

or putting a rule in our database like,

```
h(X):-g1(X),g2(Y).
```

we would enter the auxiliary rule,

```
r1(Y):-g2(Y),!.
```

and then evaluate the equivalent query,

```
bagof(X,(g1(X),r1(Y)),B).
```

or put the following equivalent rule into our database,

```
h(X):-g1(X),r1(Y).
```

The latter query and rule will yield the same results but without all the unecessary backtracking and inference. The body of the auxiliary rule is appropriately evaluated as an "independent subproblem." The savings in resource use can obviously be enormous.

Extending this sort of treatment to more complicated queries and rules is not trivial, but not terribly hard either. Consider the following sort of case, for example,

```
h(X,Z):-g1(X),g2(Y),g3(Z).
```

In this case we do not want to enter the auxiliary rule,

```
r1(Y,Z):-g2(Y),g3(Z),!.
```

and change our original rule to,

```
h(X,Z):-g1(X),r1(Y,Z).
```

since this procedure would only allow us to find one of the possibly many solutions to g3(Z). The moral of this sort of case is that no head variable should occur uninstantiated in a subproblem when that subproblem is evaluated. Thus, although "g3(Z)" should not be included in a subproblem in this last example, it could be included in a subproblem in

```
h(X,Z):-g1(X,Z),g2(Y),g3(Z).
```

In this case the mentioned auxiliary rule would be appropriate, since the head variable "Z" will always be instantiated at the time "g3(Z)" is evaluated, and so its occurrence in an independent subproblem will not restrict the number of solutions found.

Another sort of case that can arise is that we may have subproblems within subproblems. Consider for example the query,

```
h(W):-g1(X)-g2(X,Y),g3(X,Z).
```

None of these goals contain head variables, so they can immediately be put into an independent subproblem. After the first of these goals has been solved, though, the remaining two goals do not share any variables, so they break into two further subproblems. Accordingly, the rule would be handled by transforming it into,

```
h(W):-r1(X,Y,Z).
```

and then we enter the following auxiliary rules,

```
r1(X,Y,Z):-g1(X),r2(X,Y),r3(X,Z),!.
r2(X,Y):-g2(X,Y),!.
r3(X,Z):-g3(X,Z),!.
```

The rationale for doing this is just the same as above. Suppose for example, that for some choice of "X" we are unable to prove "g3(X,Z)". There is no point in backtracking to find other solutions to "g2(X,Y)", since the choice of "Y" is irrelevant to our problems with "g3(X,Z)". What we need to do

is immediately go back to find another choice of "X". This is precisely what our new rules will accomplish.

This grouping of goals into subproblems is sometimes going to interact with our goal selection strategy. For example, after ordering the goals on the basis of solution cost, it may turn out that an independent subproblem is broken up by a goal containing a head variable. This sort of conflict is resolved with an optimizing algorithm which integrates the cost planning and the selective backtracking strategies we have described.

Optimizing. The optimizing algorithm that was implemented is roughly the following:

Given a rule of the form H:-G1,G2,...Gn,

(1) Order the list of goals, G1, G2,..., Gn, according to solution cost, as discussed above.

(2) Look through the goals, in order, to find head variables.

(i) If such a a goal is found, it will be the cheapest goal containing a head variable, so move it to the front of the list of goals, and assume for the remainder of the optimizing process that its arguments are instantiated. (Some of them may occur in other goals.) Consider only the remaining goals for the rest of the optimizing process. Reorder these goals according to cost, and repeat step (2).

(ii) If no such goal containing head variables is found, proceed to the next step.

(3) Any goals that remain to be considered at this point will not have any head variables at the time they are to be solved, so they constitute independent subproblems. Take the first goal Gi on the list - it will be the cheapest - and check the following goal to see if it shares any variables with Gi. If it does, it is to be included in the same subproblem with Gi, and check the next goal to see if it contains any of the same variables as Gi, and so on until there are no more goals or until a goal with no variables in common with Gi are found. At this point we have a list of the goals in the Gi subproblem, and possibly also a list of remaining goals not in the Gi subproblem. Now enter an auxiliary rule, "the Gi rule," into the database. The Gi rule is given a unique head predicate and has as head arguments all the variables that occur in the goals of the subproblem. The body of the Gi rule consists of the goals in the Gi subproblem. We now want to optimize the body of this rule as well, so assume for the remainder of the optimizing process that the variables in Gi are all instantiated. Reorder the rest of the goals in the body of Gi rule (if any) and perform this step (3) again on these goals to

find subsubproblems. Finally, reorder the list of goals outside of the Gi subproblem and perform this step (3) on them as well.

This algorithm anticipates the instantiation of variables both in its cost calculations and in its recognition of independent subproblems. It appears to be a very expensive process, but it need only be done once for any rule being put into the database, and it can actually save an enormous amount of time.

Suppose that our database contains one ground clause with the predicate "g1", one hundred ground clauses with the predicate "g2", five hundred ground clauses with the predicate "g3", and nothing else except the following definition of the predicate "h":

```
h(X):-g3(Y),g2(Z,Y),g1(X).
```

Now consider the query,

```
setof(X,h(X),S).
```

This query is obviously maximally inefficient, but our database is not really huge and so it may not be obvious that it would be worth optimizing the rule for "h(X)". The actual processing times are as follows. Executing the maximally inefficient query in the situation described takes 2291 ms. Optimizing the rule for "h" tranforms it into,

```
h(X):-g1(X),r1(Y,Z).
```

and enters the auxiliary rules,

```
r1(Y,Z):-g2(Y,Z),r2(Z),!.
r2(Z):-g3(Z),!.
```

This optimizing process takes about 280 ms. And executing the same "setof" query, but now with the optimized definition of "h" and the auxiliary rules, takes about 30 ms. Obviously, the optimizing is worthwhile in any case like this one. On a larger database, the improvements are even more dramatic, as would be expected. The optimizing code could also be compiled to improve its efficiency further once it has been put in the form in which we want to use it in any particular application.

Conclusion. The work that has been described here is aimed at providing the basis for a feasible, pragmatic deductive inference system. It is completely general and portable. The applications that this work is specifically designed for are those in which a user wants to have interactive deductive access to a database which may include general rules (expressions containing logical variables) as well as particular facts (expressions containing no variables). This sort of application would go substantially beyond most previous logic programming projects which usually require that the database contain only ground clauses or that the user cannot add new rules. It is precisely the more general sort of database system that exploits the real advantages of a deductive system, though, and this sort of system would be required in many question answering systems.

References.

Chang, C. and Lee, R.C. (1973) Symbolic Logic and Mechanical Theorem Proving. New York: Academic Press.

Clark, K. L. and McCabe, F. (1982) IC-PROLOG - language features. In K.L. Clark and S.-A. Tarnlund, eds., Logic Programming. New York: Academic Press.

Clocksin, W.F. and Mellish, C.S. (1981) Programming in PROLOG. Berlin: Springer-Verlag.

Elcock, E.W. (1982) Goal selection strategies in Horn clause programming. Proceedings of the Fourth National Conference of the Canadian Society for Study in Artificial Intelligence.

Elcock, E.W. (1983) The pragmatics of PROLOG - some comments. Unversity of Western Ontario, Department of Computer Science Technical Report.

Elcock, E.W., Stabler, E.P., Wyatt, D., and Young, A. (forthcoming) Database management in PROLOG. Unpublished technical report.

Henschen, L. and Wos, L. (1974) Unit refutations and Horn sets. JACM, 21, pp 590-605.

Hermes, H. (1965) Enumerability, Decidability, Computability. New York: Springer-Verlag.

Moore, R.C. (1980) Reasoning about Knowledge and Action. SRI Technical Note 191.

Moore, R.C. (forthcoming) The role of logic in knowledge representation and commonsense reasoning.

Pereira, L.M.; Pereira, F.C.N. and Warren, D.H.D. (1978) User's Guide to DECsystem-10 PROLOG.

Stabler, E.P. (1982) Database and theorem prover designs for question answering systems. Centre for Cognitive Science technical report, Cogmem No. 12, University of Western Ontario.

Warren, D.H.D. (1981) Efficient processing of interactive relational database queries expressed in logic. Department of Artifical Intelligence Research Paper No. 156, University of Edinburgh.

Warren, D.H.D. and Periera, F.C.N. (1981) An efficient easily adaptable system for interpreting natural language queries. Department of Artificial Intelligence Research Paper No. 155, University of Edinburgh.