NEGATION AS FAILURE

Keith L. Clark

Department of Computer Science & Statistics

Queen Mary College, London, England

ABSTRACT

A query evaluation process for a logic data base comprising a set of clauses is described. It is essentially a Horn clause theorem prover augmented with a special inference rule for dealing with negation. This is the negation as failure inference rule whereby ~ P can be inferred if every possible proof of P fails. The chief advantage of the query evaluator described is the effeciency with which it can be implemented. Moreover, we show that the negation as failure rule only allows us to conclude negated facts that could be inferred from the axioms of the completed data base, a data base of relation definitions and equality schemas that we consider is implicitly given by the data base of clauses. We also show that when the clause data base and the queries satisfy certain constraints, which still leaves us with a data base more general than a conventional relational data base, the query evaluation process will find every answer that is a logical consequence of the completed data base.

INTRODUCTION

Following Kowalski [1978] and van Emden [1978] we consider a logic data base to comprise a set of clauses. The ground unit clauses are the extensional component of the data base. They provide us with a set of instances of the data base relations. The remaining clauses constitute the intensional component, they are the general rules of the data base. The general rules as well as the explicit 'data' are to be used in the deductive retrieval of information.

The shortcoming of such a data base is that, like a more conventional relational data base (Codd [1970]), it only contains information about true instances of relations. Even so, quite straightforward queries make use of negation, and can be answered only by showing that certain relation instances are false. Thus, to answer a request for the name of a student not taking a particular course, C, we need to find a student, S, such that Takes(S,C) is false. For his relational calculus (Codd [1972]), Codd's solution to the problem is to assume that a tuple is in the complement of the relation if it is not a given instance of the relation. For a logic data base, where a relation instance which is not explicitly given may still be implied by a general rule, the analogous assumption is that a relation instance is false if we fail to prove that it is true. The great advantage of such a 'solution to negation' is the ease with which it can be implemented. To show that P is false we do an exhaustive search for a proof of P. If every possible proof fails, ~ P is 'inferred'. This is the way that both PLANNER (Hewitt [1972]) and PROLOG (Roussel [1975], Warren et al. [1977]) handle negation.

What we have here is a proof rule:

$\vdash \sim \vdash P$ infer $\vdash \sim P$

where the proof that P is not provable is always the exhaustive but unsuccessful search for a proof of P. Let us call it the negation as failure inference rule. For pragmatic reasons this is adopted as the sole inference rule for negated formulae. Is the consequence that we have given "~" a new meaning as "fail to prove", or can we perhaps reconcile negation, operationally understood as "fail to prove", with its truth functional semantics? In other words, can we interpret a failure proof of ~ P as a valid first order inference that P is false.

Note that to assume that a relation instance is false if it is not implied, is to assume that the data base gives complete information about the true instances of its relations. This is the closed world assumption referred to by Reiter [1978] and Nicolas and Gallaire [1978]. More precisely, it is the assumption that a relation instance is true only if it is given explicitly or else is implied by one of the general rules for the relation. Thus, let us suppose that the data base contains just two instances of the unary relation, Maths-course:

Maths-course(C101)←
Maths-course(C301)← (1)

and no general rules about this relation. For any course name C, different from C101 and C301, Maths-course(C) is not provable. But to conclude, in consequence, that Maths-course(C) is false, is to assume that C101 and C301 are the only Maths-courses, and that C is not an alternative name for either course. If we were to make these assumptions explicit we would need to add to our data base the completion law:

$$(\forall x)[\text{Maths-course}(x) \rightarrow x=C101 \vee x=C301] \qquad (2)$$

and the inequality schemas:

$C101 \neq C$	for all names C different from C101
$C301 \neq C$	for all names C different from C301

Note that (1) and (2) are together equivalent to a definition

$$(\forall x)[\text{Maths-course}(x) \leftrightarrow x=C101 \vee x=C301]$$

of the Maths-course relation.

As we might expect, every negated fact ~ Maths-course(C) that we can 'infer' by showing that Maths-course(C) is not given, can now be proved by a first order deduction using the completion law and inequality schemas. More than that, the failure proof of Maths-course(C) and the first order deduction are structurally almost identical (see Figure 1). The alternatives of the failed proof space become explicit disjunctions in the first order deduction, match failures of the former become <u>false</u> equalities of the latter. Thus the failure proof is essentially a structural representation of a natural deduction style proof:

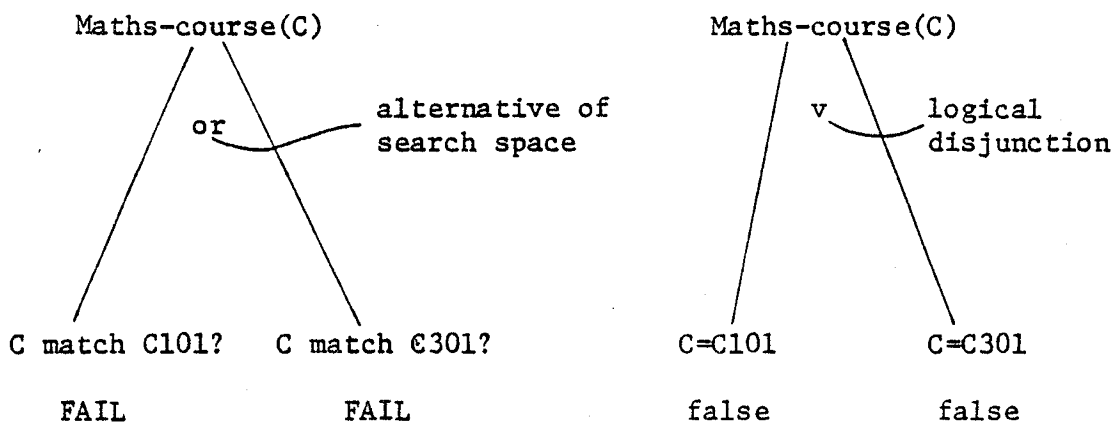


Figure 1. First Order Deduction

$$
\begin{aligned}
\text{Maths-course}(C) &\leftrightarrow C{=}C101 \vee C{=}C301 \\
&\leftrightarrow \text{false} \vee \text{false} \\
&\leftrightarrow \text{false} \\
\therefore\ &\sim \text{Maths-course}(C)
\end{aligned}
$$

This suggests a way of reconciling negation as failure with its truth functional semantics. We can assume that the clauses that appear in a logic data base B comprise just the if-halves of a set of if-and-only-if definitions of the data base relations, the only-if half of each definition being a completion law for the relation. The completed data base C(B), implicitly given by the clauses of B, comprises this set of definitions together with a set of equality schemas which make explicit the convention that different object names denote different objects. If we can show that every failed attempt to prove P using just the data base of clauses B, is in effect a proof of ~ P using the completed data base C(B), then 'negation as failure' is just a derived inference rule for deductions from C(B).

In this paper we present just such a validation of the negation as failure inference rule. The structure of the paper is as follows. In the next section, Query Evaluation for a Data Base of Clauses, we make precise what constitutes a data base of clauses, a data base query, and the deduction process of query evaluation. As we describe it, query evaluation is a non-deterministic process for which every evaluation path either succeeds, or fails, or does not terminate. A negated literal ~ P is evaluated by recursively entering the query evaluator with the query P. If every possible evaluation path for P ends in failure, we return with ~ P evaluated as true. These recursively constructed failure proofs can be nested to any depth.

In the section, Data Base Completion, we define the completion of a data base of clauses. Then in the section, Correctness of Query Evaluation, we give the formal results validating the use of the negation as failure inference rule. We prove that a query evaluation will only produce results that are implied by the completed data base. The proof is constructive. It gives us a method for reformulating a query evaluation from the data base of clauses as a deduction from the completed data base. In the last section, Completeness of Query Evaluation, we address the issue of the deductive completeness of the query evaluation process. That is, we deal with the issue of whether or not there are answers to a query implied by the completed data base which are not the answer of any evaluation of the query.

As a final word of introduction we should say something about the relationship between the negation as failure inference rule treated here and the deduction rules considered by Kramosil [1975]. Kramosil shows that the adoption of sound deduction rules of the form

$$\text{from} \quad \vdash A, \ \sim\vdash B \quad \text{infer} \quad \vdash C$$

which contain preconditions such as $\sim\vdash B$ about unprovability of certain formula, cannot extend the class of theorems that are derivable in a first order theory. There are two differences between the negation as failure rule

$$\text{from} \ \vdash \sim\vdash P \quad \text{infer} \ \vdash \sim P$$

and the rules to which Kramosil's result applies.

The first difference, which is not the major difference, is that the single rule that we consider has a definite and proscribed method for proving unprovability. It is, in fact, a relatively weak rule, for the unprovability condition is so strong. Kramosil leaves unspecified the means by which the unprovability of a formula would be determined.

The second, and crucial difference, is that his formal result applies only to inference rules that sit on top of a complete inference system for first order logic. However, the negation as failure inference rule sits on top of a resolution inference procedure which, without the rule, can only cope with Horn clause refutations. In other words, it is used to extend the deduction power of inference system that is <u>not</u> a complete deductive system for first order logic. We show that the enrichment of a Horn clause theorem prover with this inference rule is, at least when seeking answers to a query logically implied by the axioms of a completed data base, somewhat of an alternative to using a more conventional deductive system which is not restricted to Horn clauses.

QUERY EVALUATION FOR A DATA BASE OF CLAUSES

We assume a familiarity with the terminology of resolution logic (but see Chang and Lee [1973] for an introduction). The statements of the data base are a set of clauses each of which contains a distinguished positive literal. The relation of this literal is the relation that the clause is <u>about</u>. Using the notation of Kowalski [1978], the clauses are written as implications of the form

$$R(t_1,..,t_n) \leftarrow L_1 \& L_2 \& \ldots \& L_m \ , \ m \geq 0 \qquad (3)$$

Here, $R(t_1,\dots,t_n)$ is the distinguished positive literal (so the clause is <u>about</u> the relation R), the $L_1,\dots,L_m$ are all literals, and each free variable is implicitly universally quantified over the entire implication. In more conventional clause notation this would be written as the disjunction

$$R(t_1,\dots,t_n) \vee \sim L_1 \vee \sim L_2 \vee \dots \vee \sim L_m \qquad (4)$$

Note that any other positive literal of the disjunctive form (4) will appear as a negated precondition of the implication (3). When m=0, we have a unit clause.

Example Data Base

Table 1 gives the clauses of a micro logic data base. The unit clauses, which are all ground, are the explicitly given relation instances. The single non-unit clause is a general rule for the relation Non-maths-major. Since the variable y appears only in the antecedent of this clause we can read it as

> Anything x is a Non-maths-major if there is a Maths-course y which x does not take.

The functor "." is simply a data constructor, constructing a compound name "J.Brown" from the initial "J" and the surname "Brown". Every functor of a logic data base has this data constructor role. In logic parlance, the functors implicitly have their free or Herbrand interpretation. In our example data base, if we required to refer to the surname or initial of a name separately, we would include the general rules:

Initial(x.y,x)←
Surname(x.y,y)←

Table 1. Micro Data Base

Student(J.Brown)←	
	Takes(J.Brown,C101)←
Student(D.Smith)←	
	Takes(D.Smith,C101)←
	Takes(D.Smith,C301)←
Maths-course(C101)←	
Maths-course(C301)←	
Non-maths-major(x) ← Maths-course(y) & ~ Takes(x,y)	

Queries

A query is a conjunction of literals written as

$\leftarrow L_1 \& \ldots \& L_n$

If $x_1, \ldots, x_k$ are the variables appearing in the query we interpret this as a request for a constructive proof that

$(\exists x_1, \ldots, x_k)\ L_1 \& \ldots \& L_n$;

constructive in the sense that the proof should find a substitution

$\theta = \{x_1/e_1, x_2/e_2, \ldots, x_k/e_k\}$

such that

$(L_1 \& \ldots \& L_n)\theta$

is a logical consequence of the completed data base. θ is an answer to the query. For the time being we ignore the more general query which asks for the set of all answers.

Example Queries

The following are queries for the logic data base of Table 1.

(i) $\leftarrow$Student(x) & Takes(x,C101)

(ii) $\leftarrow$Student(x) & Non-maths-major(x)

(iii) $\leftarrow$Student(x) & $\sim$ Non-maths-major(x)

The first is a request for the name of a student who takes course C101; the second is a request for a student who is a non-maths-major, for a student who does not take all the maths courses; in contrast the third is a request for a student who is not a non-maths-major, for a student who does take all the maths courses. In the relational calculus, or a more elaborate query notation, such a condition on the data to be retrieved would be expressed by the formula

(iv) $\leftarrow$Student(x) & $\forall$y[Maths-course(y) $\rightarrow$ Takes(x,y)]

in which the free variable x refers to the data to be retrieved. There is no reason why the user query language should not allow such general form queries. For, by the simple expedient of introducing clauses about auxiliary relations, we can translate such a query into our standard form of a conjunction of literals. Thus, the expression of query (iv) is equivalent to

$\leftarrow$Student(x) & $\sim\exists$y [Maths-course(y) & $\sim$ Takes(x,y)]

which is equivalent to

$\leftarrow$Student(x) & $\sim$ Non-maths-major(x)

where

Non-maths-major(x) $\leftrightarrow$ $\exists$y[Maths-course(y) & $\sim$ Takes(x,y)]

The if-halve of the definition of Non-maths-major is the clause about the relation that we included in the micro data base of Table 1. It was because of this that we were able to express query (iv) directly as the standard form query (iii). As we shall see, the presence of this single clause about Non-maths-major is tantamount to giving it to the above definition in the completed data base.

The Query Evaluation Process

The query evaluation process is essentially a linear resolution proof procedure with negated literals 'evaluated' by a failure proof. However we shall view the alternate derivations of the search space as different paths of a non-deterministic evaluation which can SUCCEED, FAIL or not terminate. A path terminates with SUCCESS if its terminal query is the empty query. The binding of the variables of the initial query induced by a successful evaluation is an answer to the query. A path terminates with FAILURE if the selected literal of its terminal query does not unify with the consequent literal of the selected data base clause. The literal is selected using a prescribed selection rule, but the subsequent selection of a data base clause, and the attempted unification, is a non-deterministic step of the evaluation. Finally, a non-terminating evaluation path comprises an infinite sequence of queries each of which is derived from the initial query as described below.

Evaluation Algorithm

Until an empty query is derived, and the evaluation succeeds, proceed as follows:

Using the selection rule, select a literal L_i from the current query $\leftarrow L_1 \& \ldots \& L_n$. The selection rule is constrained so that a negative literal is only selected if it contains no variables.

Case 1.

L_i is a positive literal $R(t_1, \ldots, t_n)$

Non-deterministically choose a database clause

$$R(t'_1,\ldots,t'_n) \leftarrow L'_1\&\ldots\&L'_m$$

about R and try to unify L_i, with $R(t'_1,\ldots,t'_n)$. If there are several data base clauses about R, the selection of a clause together with the attempted unification is a non-deterministic step in the evaluation. Each of the other clauses offer an alternative evaluation path. If L_i does not unify with $R(t'_1,\ldots,t'_n)$, FAIL (this path). If L_i does unify, with most general unifier θ, replace the current query with the derived query

$$\leftarrow\{L_1\&\ldots\&L_{i-1}\ \&\ L'_1\&\ldots\&L'_m\ \&\ L_{i+1}\&\ldots\&L_n\}\theta$$

Should there be no data base clauses about the relation of the selected literal, we consider that there is just one next step to the evaluation of the query

$$\leftarrow L_1\&\ldots\&L_n$$

which immediately FAILS.

<u>Case 2.</u>

L_i is a negative ground literal $\sim P$. There is just one next step for the evaluation. This is the attempt to discover whether $\sim P$ can be assumed as a lemma. To do this we recursively enter the query evaluation process with $\leftarrow P$ as a query.

If the evaluation of $\leftarrow P$ SUCCEEDS, FAIL.

If the evaluation of $\leftarrow P$ FAILS for every path of its nondeterministic evaluation, assume $\sim P$ as a lemma. Hence replace the current query by

$$L_1\&\ldots\&L_{i-1}\ \&L_{i+1}\&\ldots\&L_n\ .$$

Remarks

(1) Note that in the special case that no negative literals appear in the query evaluation what we have described is essentially LUSH resolution (Hill [1974]), a linear resolution inference procedure for Horn clauses.

(2) Only the distinguished positive literal of each clause of the data base is a candidate for unification with a query literal. Other positive literals of the clause (that appear as negated preconditions of the implication) are never resolved upon; they can only be deleted after a failure proof.

(3) The different possible selections of a literal in a query <u>do not</u> provide alternatives for the evaluation process. However any

rule for selecting the literal can be used. The PROLOG selection rule is - always choose the leftmost literal in a query.

(4) The constraint that a negative literal should only be selected if it is ground is not a significant restriction. It just means that every variable appearing in a negated condition should have its 'range' specified by some unnegated condition. This is just the constraint that Codd imposes on the use of negation in his relational calculus.

(5) Let $\theta_1, \ldots, \theta_n$ be the sequence of unifying substitutions of a successful evaluation of some query Q. Let θ be the composition

$$\theta_1 \circ \theta_2 \circ \ldots \circ \theta_n$$

of these unifying substitutions. The subset of θ which gives the substitutions for variables of the query Q, augmented with the identity substitution for any variables of Q not bound by θ , is the answer given by the evaluation. If Q has no variables, the answer is <u>true</u>. On the other hand, if every evaluation path of the query Q ends with FAIL, the answer is <u>false</u>.

(6) Suppose that for some selection rule every branch of the evaluation tree rooted at a query Q terminates with a SUCCESS or FAIL. By König's Lemma (see Knuth [1968]), the evaluation tree contains a finite number of queries. Hence, by back-tracking when the non-deterministic evaluation succeeds or fails, we can find every answer of the evaluation tree. In the section, Completeness of Query Evaluation, we shall consider constraints which guarantee finite evaluation trees.

(7) Using Boyer and Moore [1972] structure sharing methods a back-tracking search for a successful evaluation can be achieved by manipulating a stack of activation records as in a more conventional computation. Broadly speaking, the currently derived query is represented by the entire stack. The i'th activation record on the stack records the subset of literals that were introduced at the i'th resolution step but which have not yet been deleted by a subsequent resolution. These literals are represented implicitly by a pointer to the data base clause that was used and a tuple of substitutions - the binding environment for this activation of the data base clause. The activation record also contains information about the data base clauses that have not yet been used to resolve on its literals. This information is used for back-tracking. Should one of these literals have a relation R that is extensionally characterised by a large set of unit clauses, the back-tracking information might be a pointer into a file of R-records. In which case a back-tracking search for a clause that matches the literal is just a search through the file. We can of course index the file to speed up the search. This is a very brief and slightly simpli-

fied description of the implementation possibilities. For more details the reader can consult Warren et al. [1977]. However, with the query evaluation process implemented using such techniques we can justly claim that its execution is a computational retrieval of information.

DATA BASE COMPLETION

Remember we are going to validate the query evaluation process as an inference not from the data base of clauses, but from the completed data base, the data base of definitions and equality schemas implicitly given by the set of clauses.

Suppose that

$$R(t_1,\ldots,t_n) \leftarrow L_1\&\ldots\&L_m \tag{5}$$

is a data base clause about relation R. Where = is the equality relation, and $x_1,\ldots,x_n$ are variables not appearing in the clause, it is equivalent to the clause

$$R(x_1,\ldots,x_n) \leftarrow x_1=t_1\&\ldots\&x_n=t_n\&L_1\&\ldots\&L_m$$

Finally, if $y_1,\ldots,y_p$ are the variables of (5), this is itself equivalent to

$$R(x_1,\ldots,x_n) \leftarrow (\exists y_1,\ldots,y_p)[x_1=t_1\&x_2=t_2\&\ldots\&x_n=t_n\&L_1\&\ldots\&L_m] \tag{6}$$

We call this the <u>general form of the clause</u>.

Suppose there are exactly k clauses, $k>0$, in the data base about the relation R. Let

$$\begin{aligned} &R(x_1,\ldots,x_n) \leftarrow E_1 \\ &\qquad\qquad\vdots \\ &R(x_1,\ldots,x_n) \leftarrow E_k \end{aligned} \tag{7}$$

be the k general forms of these clauses. Each of the E_i will be an existentially quantified conjunction of literals as in (6). The <u>definition of R</u>, implicitly given by the data base, is

$$(\forall x_1,\ldots,x_n)[R(x_1,\ldots,x_n) \leftrightarrow E_1 \vee E_2 \vee \ldots \vee E_k]$$

The if-half of this definition is just the k general form clauses (7) grouped as a single implication. The only-if half is the completion law for R.

Should there be no data base clauses for R, the definition implicitly given by the data base, is

$$(\forall x_1,\dots,x_n)[R(x_1,\dots,x_n) \leftrightarrow \text{false}]$$

<u>Example</u>

Suppose

$P(a)\leftarrow$

$P(b)\leftarrow$

$P(f(y)) \leftarrow P(y)$

are all the clauses about a unary relation P. Its disjunctive definition is

$$(\forall x)[P(x) \leftrightarrow x{=}a \lor x{=}b \lor \exists y[x{=}f(y) \& P(y)]] \blacksquare$$

In moving to the disjunctive definitions from the original clauses we have been forced to introduce equalities. Thus the onus is upon us to say something about the equality relation for the objects of the data base. That is we need to state explicitly that the constants and functors have their free interpretation. The following schemas suffice. Each schema is implicitly universally quantified with respect to its variables.

$c \neq c'$ c,c' any pair of distinct constants (8)

$f(x_1,\dots,x_n) \neq g(y_1,\dots,y_m)$ f,g any pair of distinct functors (9)

$f(x_1,\dots,x_n) = f(y_1,\dots,y_n) \rightarrow x_1{=}y_1 \& \dots \& x_n{=}y_n$ f any functor (10)

$f(x_1,\dots,x_n) \neq c$ f any functor, c any constant (11)

$\tau(x) \neq x$ $\tau(x)$ any term structure in which x is free (12)

Schema (8) tells us that different constants denote different objects. Schema (9) tells us that different functors are different data constructors, and (10) tells us that constructed objects are equal only if they are constructed from equal components. Axioms (11) and (12) together tell us that the data constructors always generate new objects.

The above axioms, together with the following general axioms for equality:

$$x = x$$

$$x = y \rightarrow y = x$$

$$x = y \ \& \ y = z \rightarrow x = z$$

Substitution schema:

$$x = y \rightarrow [W(x) \leftrightarrow W(y)], \text{ W any wff}$$

we call the identity theory for a completed data base.

The identity theory, together with the set of relation definitions implicitly given by a logic data base, constitute the completed data base.

Example.

The definitions and axioms of Table 2 are the completed data base of Table 1. In the definitions each free variable is implicitly universally quantified.

CORRECTNESS OF QUERY EVALUATION

In this section we give the formal results that validate query evaluation from a data base of clauses as a first order inference from the completed form of the data base. The main results are Theorems 2 and 3. Theorem 2 is the validation of negation as failure. It is the proof that whenever for some selection rule every branch of the query evaluation tree ends in a failure, then the

Table 2. Completed Data Base of Table 1.

$\text{Student}(x) \leftrightarrow x{=}\text{J.Brown} \lor x{=}\text{D.Smith}$

$\text{Maths-course}(x) \leftrightarrow x{=}\text{C101} \lor x{=}\text{X301}$

$\text{Takes}(x,y) \leftrightarrow x{=}\text{J.Brown} \ \& \ y{=}\text{C101} \lor x{=}\text{D.Smith} \ \& \ y{=}\text{C101}$
$\qquad\qquad\qquad\qquad\qquad\qquad \lor x{=}\text{D.Smith} \ \& \ y{=}\text{C301}$

$\text{Non-maths-major}(x) \leftrightarrow (\exists y)[\text{Maths-course}(y) \ \& \sim \text{Takes}(x,y)]$

For any other relation, a definition that it is always false.

The identity theory

construction of the tree is in effect a proof that there are no answers to the query. Theorem 3 is a generalisation of this result for an evaluation tree every branch of which terminates with a failure or a success. It tells us that the set of answers given by the success branches are provably the only answers to the query. Each of these theorems relies on Theorem 1. Roughly speaking, this tells us that a query Q is equivalent to the disjunction of the queries derivable from Q by resolving on any positive literal. This, in turn, relies on the fact that by using the identity theory we can emulate unification by inference about equalities:

Lemma.

(1) If $R(t_1,\ldots,t_n)$ unifies with $R(t_1',\ldots,t_n')$ with m.g.u.

$$\theta = \{x_1/e_1,\ldots,x_k/e_k\}$$

then, using the identity theory of the completed data base, the conjunction of equalities

$$t_1=t_1' \&\ldots\& \ t_n=t_n'$$

is provably equivalent to

$$x_1=e_k \ \&\ldots\& \ x_k=e_k$$

(2) If $R(t_1,\ldots,t_n)$ does not unify with $R(t_1',\ldots,t_n')$ then, using the identity theory,

$$t_1=t_1' \ \&\ldots\& \ t_n=t_n'$$

is provably equivalent to false.

Proof

For brevity we simply sketch the proof. What is needed is an induction on the number of steps of an attempt to unify the two literals. Schema (10) of the identity theory is used to infer e=e' where e and e' are corresponding sub-terms which differ. If e is a variable and e' a term in which the variable does not appear, we use the substitution schema for equality to 'apply' the substitution {e/e'}. Otherwise, one of the inequalities (8), (9), (11) or (12) gives us the contradiction. ■

Our first theorem is a direct application of this lemma and the fact that the completed data base defines a relation as the disjunction of antecedents of its general form clauses. To state it we need the concept of the general form of a derived query.

Definition.

Let Q' be the query derived from some query Q by selecting a positive literal $R(t_1,\ldots,t_n)$ and resolving with a data base clause C. The general form of the resolvent Q' is

$$(\exists y_1,\ldots,y_p)[z_1=e_1\&\ldots\&z_k=e_k \;\&\; Q']$$

Where $\{z_1/e_1,\ldots,z_k/e_k\}$ is the subset of the m.g.u. θ which applies to variables of Q, and $y_1,\ldots,y_p$ are the variables of data base clause C that remain in $e_1,\ldots,e_k$ or Q'. ■

Theorem 1

Let Q be a query which contains the positive literal $R(t_1,\ldots,t_n)$. Suppose $Q_1,\ldots,Q_j$ are all the alternative queries that are derivable from Q by resolving on $R(t_1,\ldots,t_n)$ with some data base clause about R. If $G_1,\ldots,G_j$ are the general forms of these derived queries

$$Q \leftrightarrow G_1 \vee G_2 \vee \ldots \vee G_j$$

is a theorem of the completed data base. If j=0, i.e., there are no queries derivable from Q by resolving on $R(t_1,\ldots,t_n)$, then

$$Q \leftrightarrow \text{false}$$

is a theorem of the completed data base. ■

The proof of this theorem is quite straightforward. We are now in a position to establish our first main result. We want to show that the construction of a failure evaluation tree - an evaluation tree every branch of which terminates with a failure - is tantamount to a first order proof that the root query has no solutions.

Figure 2 gives the structure of such a failure evaluation tree. Every branch of the tree, every evaluation path, ends at a terminal query Q' whose off-springs are all failure nodes. We want to prove that whenever some query Q is the root of a failed evaluation tree

$$Q \leftrightarrow \text{false}$$

or, equivalently

$$\sim (\exists z_1,\ldots,z_n)Q$$

where $z_1,\ldots,z_n$ are the free variables of Q, is a theorem of the completed data base.

As a special case let us consider a failure tree T which records the top-level structure of a failure proof that does not depend on any subsidiary failure proofs. For such a failure tree the selected literal of the root query Q must be a positive literal $R(t_1,\ldots,t_n)$, with $Q_1,\ldots,Q_j$ the set of resolvents with the data base clauses about R. But Theorem 1 tells us that

$$Q \leftrightarrow G_1 \vee .. \vee G_j$$

where $G_1,\ldots,G_j$ are the general forms of $Q_1,\ldots,Q_j$. Clearly, if each of the Q_i is provably equivalent to false so is its general form. What we need therefore is an induction of the structure of T.

The above use of Theorem 1 is our induction step. The base case of the induction is established by considering the two single query failure trees of Figure 3 and Figure 4.

Theorem 1 again covers the case of Figure 3. When T is as depicted in Figure 4 it records a failure proof of the root query because the recursively entered evaluation of $\leftarrow P$ has succeeded. But in this case the successful evaluation of $\leftarrow P$ will be a resolution proof of P. It would be other than a straightforward resolution proof only if it involved the failure proof of some negated literal. However we have discounted this possibility by our assumption that the failure tree records a failure proof that does not

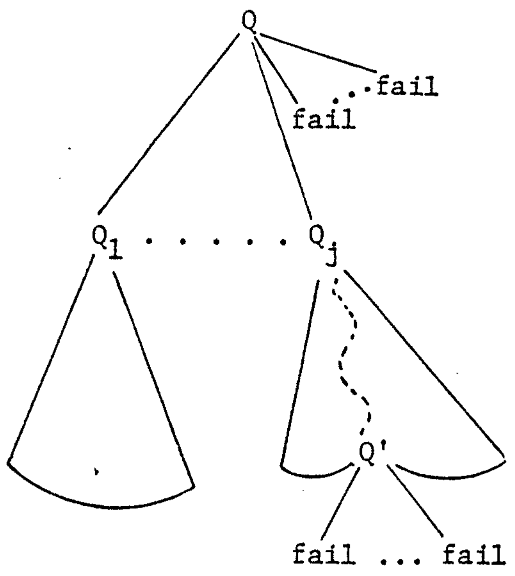
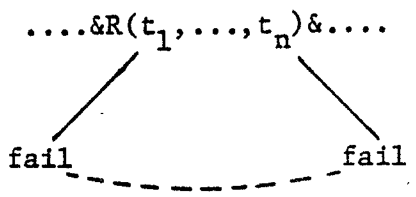


Figure 2. Failure Evaluation Tree



R(t_1,...,t_n) fails to unify with any data base clause about R.

Figure 3. Single Failure Tree

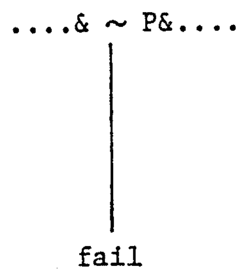
depend on any subsidiary failure proofs. Moreover, a resolution proof of P from the data base of clauses is also a proof that

$$\sim P \leftrightarrow \text{false}$$

from the completed data base. Hence the root query is again provable equivalent to false.

We now need to establish the result for a failure tree T whose construction may depend on auxiliary failure proofs. Since the number of such auxiliary proofs must be finite, we do this by an induction on the number, n, of these auxiliary failure proofs.

The above argument establishes the base case, n=0, of this induction. Let us now assume that the root query of a failure tree whose construction depends on less than n subsidiary failure proofs is provably equivalent to false. Let T be a failure tree whose construction depends on at most n failure proofs $n > 0$.



A query evaluation for ←P succeeds.

P is a ground literal

Figure 4. Single Failure Tree

As above, we show that the root query of T is provably equivalent to false by a secondary induction of the structure of T. Again the base cases of this structural induction are as depicted in Figures 3 and 4. As before, Theorem 1 covers the case of Figure 3. However, this time the single query tree of Figure 4 records a successful evaluation of $\leftarrow P$ which may depend on a failure proof of some negated literal $\sim L$, this failure proof being the construction of a failure tree rooted at L. But such a failure proof can itself make use of at most n-1 subsidiary failure proofs. By the induction hypothesis

$$L \leftrightarrow \text{false}$$

and hence

$$\sim L$$

is a theorem of the completed data base. This applies to any failure proved negated literal selected in the evaluation of $\leftarrow P$. The deletion of such a negated literal can therefore be regarded as a resolution step which uses a lemma $\sim L$ of the completed data base. Since every other step in the evaluation of $\leftarrow P$ is just a resolution with a data base clause, we have again that

$$\sim P \leftrightarrow \text{false}.$$

is a thoerem of the completed data base.

The induction step of our structural induction on T is just a slight elaboration of the argument for the special case failure tree that required no subsidiary failure proofs. This time we must also consider the failure tree structure depicted in Figure 5. Here the root query has a single off-spring derived by deleting a negated literal $\sim L$, which has been failure proved. But again the failure proof of $\sim L$

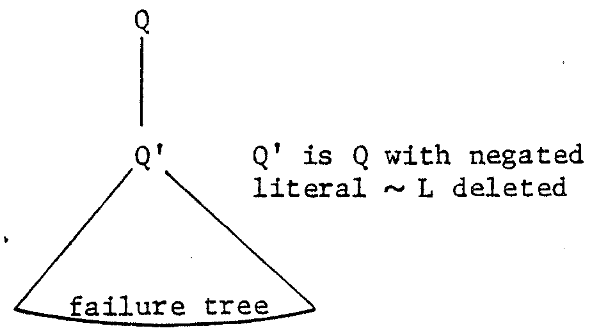


Figure 5. Failure Trees

can itself depend on at most n-1 failure proofs, so $\sim L$ is a theorem of the completed data base. Therefore

$$Q \leftrightarrow Q'$$

is also a theorem. But Q' is the root of a sub-tree which is a failure tree, so

$$Q' \leftrightarrow \text{false}$$

and hence

$$Q \leftrightarrow \text{false}$$

as required. We have proved:

Theorem 2.

If for some literal selection rule every branch of the evaluation tree of a query $\leftarrow Q$ terminates with failure, then

$$\sim(\exists x_1,\ldots,x_k)Q$$

where $x_1,\ldots,x_k$ are the free variables of Q, is a theorem of the completed data base. ■

Let us now look at the case of a successful query evaluation. By the above theorem, any failure evaluation of a negated literal $\sim L$ that it might use can be viewed as the derivation from the completed data base of a lemma $\sim L$. Thus the whole evaluation can be viewed as a linear resolution proof using the data base clauses (the if-halves of the relation definitions) and a set of negated ground literal lemmas. Suppose that the answer given by the successful evaluation is

$$\theta = \{x_1/e_1,\ldots,x_i/e_i,x_{i+1}/x_{i+1},\ldots,x_k/x_k\}$$

$x_1,\ldots,x_i$ being the subset of the free variables $x_1,\ldots,x_k$ of the query Q that are bound by the evaluation. By the soundness of resolution

$$(\forall x_{i+1},\ldots,x_k)(\forall y_1,\ldots,y_n)Q\theta \;,$$

where $y_1,\ldots,y_n$ are the extra free variables of $Q\theta$ introduced by the substitution θ , is a theorem of the completed data base.

It follows that

$$(\forall x_1,\ldots,x_i,x_{i+1},.,x_k)(\forall y_1,\ldots,y_n)[x_1=e_1\&\ldots\&x_i=e_i \rightarrow Q]$$

is also a theorem. Finally, since $y_1,..,y_n$ were introduced by θ and do not appear in Q, this is equivalent to

$$(\forall x_1,\ldots,x_k)[(\exists y_1,\ldots,y_n)(x_1=e_1\&\ldots\&x_i=e_i) \rightarrow Q]$$

Let us call the antecedent of this conditional the general form of the answer θ, and denote it by $\hat{\theta}$.

Suppose now that there are exactly j successful evaluations of the query $\leftarrow$Q with answer substitutions $\theta_1,\ldots,\theta_j$. Then we know that

$$(\forall x_1,\ldots,x_k)\ [\hat{\theta}_1 \rightarrow Q]$$
$$\vdots$$
$$(\forall x_1,\ldots,x_k)\ [\hat{\theta}_j \rightarrow Q]$$

are all theorems of the completed data base. Suppose further that every other branch of the evaluation tree (constructed using some particular selection rule) ends in a failure node. By an exhaustive search we can discover that there are no other solutions given by this evaluation tree. We should like to know no other evaluation tree would provide us with an extra solution. Assuming that the completed data base is consistent, this is guaranteed by:

Theorem 3.

If for some literal selection rule every branch of the evaluation tree of a query $\leftarrow$Q ends with a success or failure, and $\theta_1,\ldots,\theta_j$ are all the answers given by the evaluation paths that end in success, then

$$(\forall x_1,\ldots,x_k)[Q \leftrightarrow \hat{\theta}_1 \vee \hat{\theta}_2 \vee..\vee \hat{\theta}_j]$$

is a theorem of the completed data base. Here, $x_1,\ldots,x_k$ are the free variables of Q and $\hat{\theta}_1,\ldots,\hat{\theta}_j$ are the general forms of the answers. ■

The proof is a straightforward induction on the structure of the evaluation tree.

Example Application of Theorem 2.

Figure 6 is the failure tree generated by the PROLOG selection rule for the query $\leftarrow$Non-maths-major(D.Smith). The proof of Theorem 2 gives us a method for lifting this failure tree into a first order proof of

$\sim$ Non-maths-major(D.Smith).

We simply climb down the tree substituting for each query the disjunction of general forms of its immediate descendents. By Theorem 2 each substitution preserves equivalence. For FAIL we substitute false. This deduction, with some intermediary steps inserted, is :-

Non-maths-major(D.Smith)

$\leftrightarrow$ $\exists y$[Maths-course(y)& ~Takes(D.Smith,y)] by definition of Non-maths-major

$\leftrightarrow$ $\exists y$[(y=C101 $\vee$ y=C301)& ~Takes(D.Smith,y)] by definition of Maths-course

$\leftrightarrow$ $\exists y$[y=C101& ~Takes(D.Smith,y) $\vee$ y=C301& ~Takes(D.Smith,y)]

$\leftrightarrow$ ~ Takes(D.Smith,C101) $\vee$ ~ Takes(D.Smith,C301)

$\leftrightarrow$ ~ true $\vee$ ~ true

$\leftrightarrow$ false

$\therefore$ ~ Non-maths-major(D.Smith) ■

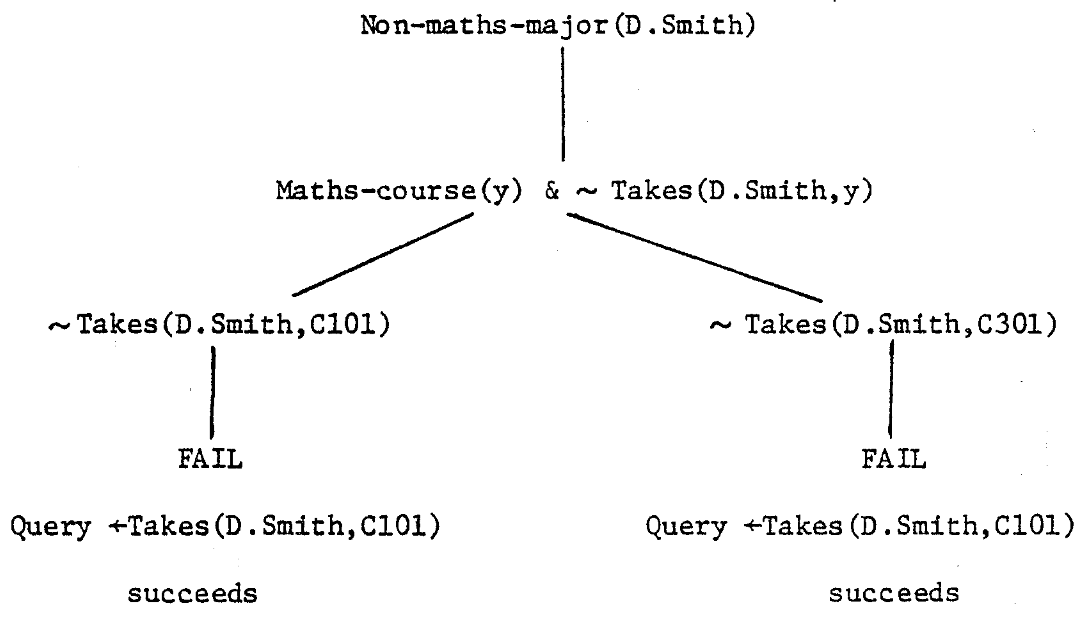


Figure 6. Failure Tree Generated by PROLOG Selection Rule

COMPLETENESS OF QUERY EVALUATION

We have argued that the evaluation of a query can and should be viewed as a deduction from the completed data base. However, when we evaluate a query, that is try to discover whether or not some instance is implied by the completed data base, we ignore the completion laws and inequality schemas. As a substitute we augment a Horn clause theorem prover for the remaining if-halves of the relation definitions - the data base clauses - with our negation as failure inference rule. Is this an adequate substitute? Will we still be able to infer every answer to the query implied by the completed data base? With certain restrictions on the data base and its queries, yes. In general, no.

Let us look at the ways in which query evaluation falls short of a complete inference system. To begin with there is the restriction that our failure inference rule should only be applied to a ground literal. With this restriction we cannot even begin to answer a query

$\leftarrow \sim R(x,a)$

which is a request for any x not related to a by R. We could relax this restriction on failure proofs. Let us suppose that the recursively entered evaluation of $\leftarrow R(x,a)$ constructs a failure tree rooted at R(x,a). By Theorem 2 we can infer $(x) \sim R(x,a)$, giving as an answer to the query the identity substitution x/x. We can also give an answer when the evaluation of $\leftarrow R(x,a)$ succeeds, providing the answer is the identity substitution x/x. For in this case the evaluation is a proof of (x)R(x,a), i.e. $\sim \exists x \sim R(x,a)$. So false is the answer to the query $\leftarrow \sim R(x,a)$. However, should we have a successful evaluation of $\leftarrow R(x,a)$ with an answer other than the identity substitution we cannot conclude anything about the query $\leftarrow \sim R(x,a)$. To patch our query evaluation process in this circumstance, we would need to resort to a systematic search for a ground instance of $\sim R(x,a)$ that can be failure proved. But of course, just such a systematic search will be invoked by the modified query,

$\leftarrow Q(x) \& \sim R(x,a)$

providing Q(x) only has ground solutions. Our insistence that queries must have this form is a requirement that the querier must implicitly constrain the search.

The second limitation associated with failure proofs is much more serious. It is the fact that we search for a failure proof by constructing just one evaluation tree.

For a query that has a successful evaluation we do not have to search alternative evaluation trees, at least not for the top-level deduction. This is because at the top-level we can view each deletion of a negated literal as a resolution step. Thus, the evaluation is in essential respects a Horn clause refutation. In the search for such a refutation we know we need only consider one selection rule (Hill [1974]). In other words, if a successful evaluation of a query Q with answer substitution θ appears on one evaluation tree rooted at Q, then it appears on every evaluation tree rooted at Q. The problem arises when we recursively enter the query evaluation process to check that some negated literal $\sim P$ is indeed a lemma of the completed data base. When we do this, to grow just one evaluation tree is to risk 'missing' a failure proof.

Suppose we have the following clauses in the data base,

$P(x) \leftarrow Q(y)\ \&\ R(y)$
$Q(h(y)) \leftarrow Q(y)$
$R(g(y)) \leftarrow$

and we want to failure prove $\sim P(a)$. If we use the PROLOG selection rule, and always select the leftmost literal, we get an evaluation tree with a single infinite branch (see Figure 7a). Using another selection rule, in fact any other rule in this instance, we get a finite failure tree (see Figure 7b).

A complete search for a failure proof should therefore search over the space of alternative evaluation trees. That is the different selections of a literal in a query should be treated as alternatives when we search for a failure proof. Interestingly, the different ways of resolving on the selected literal are not alternatives for a failure proof. Every one of them must eventually be investigated and shown to FAIL. This gives us a nice duality between search for a successful evaluation, and search for a failure proof.

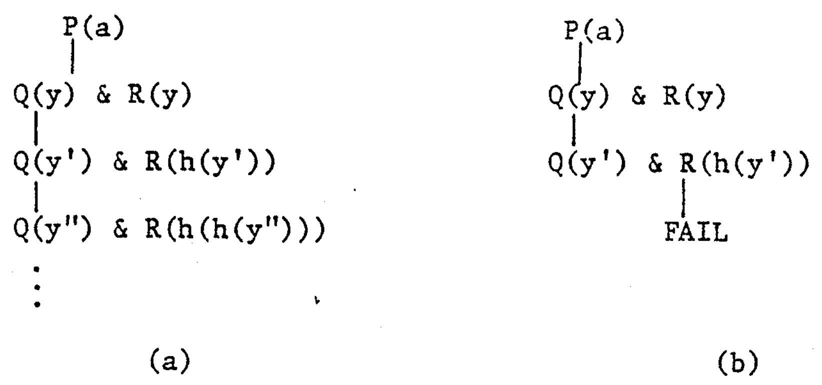


Figure 7. Incompleteness of PROLOG Selection Rule

We have seen that we might miss the failure proof of some negated literal $\sim P$ by restricting ourselves to the construction of just one evaluation tree rooted at P. In consequence, we may not be able to successfully complete a query evaluation that depends on the lemma $\sim$ P. However, this very insistence that all negated literals should be inferred as lemmas; and the consequent neglect of a case analysis proof - check if the same answer is given on the assumption that P is true, and on the assumption that P is false - is another hole in the query evaluation process.

The following clauses give an example of this:

$$\begin{aligned}
&R(x,y) \leftarrow P(x) \ \&\ Q(x,y)\\
&R(x,y) \leftarrow \sim P(x) \ \&\ T(x,y)\\
&P(x) \leftarrow P(f(x))\\
&Q(a,b)\leftarrow\\
&T(a,b)\leftarrow
\end{aligned} \tag{13}$$

R(a,b) is implied by these clauses. This is because

$$R(x,y) \leftarrow Q(x,y) \ \&\ T(x,y) \tag{14}$$

is a consequence of the two clauses for R. If we resolve these two clauses on the 'test' literal P(x) we get

$$R(x,y) \vee R(x,y') \leftarrow Q(x,y) \ \&\ T(x,y'),$$

which we can factor to give (14). Clause (14) tells us that no matter whether P(x) is true or false we can conclude R(x,y) if only Q and T can 'agree' on y. R(a,b) is now an immediate consequence of (14) and the ground clauses for Q and T. However, no evaluation of the query

$$\leftarrow R(a,b)$$

which uses only the given clauses (13) can terminate. This is because the single clause for P, which amounts to a definition

$$\forall x[P(x) \leftrightarrow P(f(x))]$$

in the completed data base, does not allow us to prove or disprove P(a), and the query evaluator insists on the proven truth or falsity of every literal encountered in the evaluation. Like intuistionist logic, the query evaluator does not countenance the law of the excluded middle

$P \vee \sim P$.

Put another way, it treats

$P \vee Q$

as a statement that is true only when $\vdash P$ or $\vdash Q$.

There is also an interesting analogy with the two different interpretations that can be given to the conditional expression

if P(x) then q(x) else t(x)

in a more conventional programming formalism. For any x, the value is g(x) if P(x) is true, and t(x) if P(x) is false. But what if P(x) is undefined, i.e., its evaluation for some argument x does not terminate, just as our evaluation of $\leftarrow$P(a) does not terminate. The 'sequential' semantics for the conditional says that the conditional is undefined. The 'parallel' semantics says that it is undefined except in the special case that g(x) and t(x) agree, i.e. return the same value. In this case the value of the conditional is this common value. This is precisely what derived clause (15) asserts. First order logic, then, insists on the 'parallel' semantics. Our query evaluation gives us the 'sequential'.

Finally, let us note that an SL refutation (Kowalski and Kuehner [1971]) used to evaluate the query would cope with the case analysis allowed by classical logic by an ancestor resolution.

Where does that leave us? In the light of the above shortcomings is query evaluation as we have described it worth considering? It is, because as we have already remarked, coupled with a back-tracking search strategy it can be most efficiently implemented. Can we perhaps side-step its inadequacies?

Firstly, the constraint that every variable in a negated literal should have its range specified by an unnegated literal that will generate a candidate set of ground substitutions is perfectly acceptable. Let us call this an allowed query. For an allowed query no evaluation can flounder because it encounters a query with only unground negative literals. (Remember the literal selection rule is constrained so that it can only select a negative literal if it is ground.) Now suppose that for some given data base we can define a literal selection rule such that the evaluation tree for every allowed query is finite. Providing the completed data base is consistent (and I think that the finiteness of every evaluation tree guarantees this, although I have not checked it out) query evaluation is complete. This is because the back-tracking traversal of the finite evaluation tree will find each and every answer given by a successful evaluation path, and Theorem 3 tells us that these are the only answers. Note that a proof that a data base + selection rule has a finite evaluation tree for each and every query is a termination proof for the data base viewed as a non-deterministic program, each posed query being a 'call' of the program.

With regards providing such a termination proof for a data base I have no ready suggestions, although I think it is an interesting area to explore. Typically we might have to modify the selection rule and perhaps further restrict the legitimate queries as a data base evolves. We can however lay down a strong but quite general condition for a logic data base which ensures termination of every allowed query evaluation for <u>any</u> selection rule. It is that each relation R of the data base, whether it be explicitly or implicitly defined, should have finite extension that can be computed by constructing any evaluation tree for the query

$$\leftarrow R(x_1,\ldots,x_k)$$

Let us call this the condition of <u>computable finite extensions</u>.

It is quite easy to show that for a data base which only has computable finite extensions the evaluation tree for an allowed query

$$\leftarrow L_1 \,\&\, L_2 \,\&\ldots\&\, L_n$$

is finite no matter what selection rule is used. Remember that any variable in a negative literal appears in some positive literal for which there are only computable ground solutions. So negative literals present no problem, all selection rules being constrained to select a negative literal only after it has become ground. We leave the reader to provide the termination proof for a query comprising only positive literals. There is a slight complication due to the fact that we can, in effect, coroutine between the evaluations of

$$\leftarrow L_1,\ \leftarrow L_2,\ldots,\ \leftarrow L_n$$

We can now come back on ourselves and use this result to specify a hierarchical data base in which each relation does have a computable finite extent, hence a data base with the termination property for any selection rule. The clauses of the data base must be such that they can be grouped into disjoint sequence of sets

$$S_0, S_1,\ldots,S_n$$

which satisfy the following condition.

Let us call a relation R of the data base in i-level relation if it is completely specified by the clauses in

$S_0 \cup S_1 \cup \ldots \cup S_i$

That is, there are no clauses about R, or any relation referred to directly or indirectly by the clauses for R, that are outside this set. The data base of clauses satisfies the hierarchical constraint if:

(i) S_0 is the set of all unit clauses which all ground

(ii) S_{i+1} only contains clauses of the form

$$L \leftarrow L_1 \,\&\, .. \,\&\, L_n$$

where the antecedent $L_1 \,\&..\&\, L_n$ is an allowed query using only j-level relations, $j \leq i$.

Note that this rules out recursive or mutually recursive definitions of relations. It is, unfortunately, a very strong constraint. It derives from the data base structuring proposals of Reiter [1977].

By an induction on i, it is easy to show that each i-level relation has a computable finite extent. Each 0-level relation is a relation completely defined by a set of ground instances. The induction step makes use of the fact that each i+1 level relation that is not also an i-level relation is defined by a set of clause each of which has a precondition

$$\leftarrow L_1 \,\&\, .. \,\&\, L_n$$

which is an allowed query about j-level relations, $j \leq i$, i.e. relations with computable finite extents. The details are quite straightforward. Another induction on i can be used to prove the consistency of the completed data base. The induction step is a proof that we can extend the model for the completion of $S_0 \cup .. \cup S_i$ to a model for the completion of $S_0 \cup \ldots \cup S_i \cup S_{i+1}$. We can conclude:

Theorem 4

For a data base satisfying the hierarchical constraint the evaluation process for allowed queries is complete. ■

The hierarchical constraint guarantees that our query evaluation process will be find each and every answer to a query. Is it too restrictive? Perhaps, but it still characterizes a data base which generalizes a conventional relational data base in the following respects:

(1) We can define the computable finite extensions of the relations by a set of instances, or by general rules, or by a mixture

of both.

(2) The components of a relation are not restricted to strings and numbers. They can be quite general data structures (terms of the logic program).

(3) The clausal notation fills the role of data description language, query language and host programming language. Indeed, a data base is just a logic program with a large number of ground clauses. This multi-role aspect of logic programs is more fully explored by van Emden [1978] and Kowalski [1978].

(4) Finally, the retrieval of information is not just a search over a set of files. It genuinely involves a computational deduction.

FINAL REMARKS

We have shown that the negation as failure inference rule applied to a data base of clauses is a sound rule for deductions from the completed data base. As a generalization of this, we have shown that an exhaustive search for solutions to a query, if it returns a finite set of solutions, is a proof that these are exactly the set of solutions. We have described a query evaluation process for a data base of clauses which uses negation as failure as its sole proof rule for negated literals. Although it is in general not complete, its chief advantage is the efficiency of its implementation. Using it the deductive retrieval of information can be regarded as a computation. However, by imposing constraints on the logic data base and its queries, which generalise the constraints of a relational data base, the query evaluation process is guaranteed to find each and every solution to a query.

ACKNOWLEDGMENTS

I have benefited much from discussions with Bob Kowalski and Maarten van Emden. The research was supported by the Science Research Council.

REFERENCES

1. Boyer, R.S. and Moore, J.S. [1972] The Sharing of Structure in Theorem Proving Programs. In *Machine Intelligence 7* (B. Meltzer and D. Michie, Eds.), Edinburgh University Press, 101-116.

2. Chang, C. L. and Lee, R.C.T. [1973] *Symbolic Logic and Mechanical Theorem Proving,* Academic Press, New York, 1973.

3. Codd, E. F. [1970] A Relational Model for Large Shared Data Banks, *CACM 13,* 6 (June, 1970), 377-387.

4. Codd, E. F. [1972] Relational Completeness of Data Base Sub-languages, In *Data Base Systems* (R. Rustin, Ed.), Prentice-Hall, 65-98.

5. Hewitt, C. [1972] Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot, *A. I. Memo No. 251,* MIT Project MAC, 1972.

6. Hill, R. [1974] Lush-Resolution and Its Completeness, *DCL Memo No. 78,* Department of Artificial Intelligence, Edinburgh University, 1974.

7. Knuth, D. [1968] *Fundamental Algorithms, The Art of Computer Programming, Vol. 1,* Addison-Wesley, Reading, Mass, 1968.

8. Kowalski, R. and Kuehner, D. [1971] Linear Resolution with Selection Function, *Artificial Intelligence 2,* 3/4 (1971), 221-260.

9. Kowalski, R. [1978] Logic for Data Description, In *Logic and Data Bases* (H. Gallaire and J. Minker, Eds.), Plenum Press, New York, N.Y., 77-103.

10. Kramosil, I. [1975] A Note on Deduction Rules with Negative Premises, *Proceedings IJCAI 4,* Tbilisi, USSR, 1975, 53-56.

11. Nicolas, J. M. and Gallaire, H. [1978] Data Bases: Theory vs. Interpretation, In *Logic and Data Bases* (H. Gallaire and J. Minker, Eds.), Plenum Press, New York, N.Y., 1978, 33-54.

12. Reiter, R. [1978] On Closed World Data Bases, In *Logic and Data Bases* (H. Gallaire and J. Minker, Eds.), Plenum Press, New York, N.Y., 1978, 55-76.

13. Reiter, R. [1977] An Approach to Deductive Question-Answering, *BBN Report No. 3649,* Bolt, Beranek and Newman, Cambridge, Mass., 1977.

14. Roussel, P. [1975] PROLOG: Manual d'Utilisation, *Rapport Interne, G.I.A., UER de LUMINY,* Universite d'Aix-Marseille, 1975.

15. van Emden, M. [1978] Computation and Deductive Information Retrieval, In *Formal Description of Programming Concepts,* (E. Neuhold, Ed.), North-Holland, 1978.