PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

DATABASE UPDATES IN PURE PROLOG

David Scott Warren

Computer Science Department
SUNY at Stony Brook
Stony Brook, NY 11794

ABSTRACT

The Prolog **assert** operator is notoriously nonlogical. By using **assert**, a programmer can write very procedural programs in Prolog, and thereby subvert one of the major advantages of programming in Prolog, the declarative nature of the resulting programs. In this paper we propose a new Prolog operator, called **assume**, which can replace many uses of **assert**. The advantage of **assume** is that it can be given a declarative semantics. Prolog programs are first-order statements, and execution of a Prolog program is deduction in the first-order logic. In an exactly analogous way, pure Prolog programs that include **assume** are statements in a modal logic, and their execution is deduction in that modal logic. The computational logic that is developed here can be understood as providing a theory of extensional updates in a relational database system. This logic provides interesting insights into the role of certain types of null values in database systems.

1. INTRODUCTION

The Prolog language is a good programming system to the extent that it is based on and reflects first-order logic. Logic developed as a declarative language. First-order statements describe states of the world, and open formulas define relationships among objects in the world. Because Prolog is an implementation of logic, statements in pure Prolog can be understood declaratively, as statements about relationships among program objects. The function of a Prolog system is to construct objects standing in the relationships defined by the program. In order to understand a program, one does not need to understand or even think about how the Prolog system finds and constructs these objects. To understand a pure Prolog program, there is no need to think about sequential computation, or the order in which things happen. This makes pure Prolog programs very easy to understand, and herein lies a power of logic programming.

The standard Prolog example of appending two lists clearly demonstrates this point.

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

(We use the now-standard DEC-10 syntax for our examples, in which variables are identifiers with initial capitals and lists are represented with square brackets.) Here **append** is a relationship among triples of lists; it is true of a triple if the third list is the concatenation of the first two. The statements above can easily be seen to be true. The first says that three lists stand in the append relationship if the first one is empty and the other two are the same list. The second statement says that three lists are in the append relationship if the first and last list have the same first element and if the tail of the first, the second, and the tail of the third stand in the append relationship. Append can be understood as simple statements about relationships among lists without any reference to how these lists might be constructed. This means that it takes much less mental effort to understand pure Prolog programs. This property of programs follows from the fact that pure Prolog is a language in which specifications are directly executable.

As an aside, this argument for the separation of what a program does from how it does it holds most strongly for readers and understanders of programs. It is certainly true that creaters of programs must consider execution details in order to be able to construct efficient programs.

It is the case, however, that most Prolog programs that are written are not pure, but use the nonlogical aspects of the Prolog programming language. Impure programs can only be understood by reference to the sequence of instructions that the interpreter executes when interpreting the program. Thus impure programs are much harder to understand and debug. The nonlogical features have been added to Prolog in order to make it into a usable programming language. Programmers seem to find that using only pure Prolog is too constraining and too inefficient. The goal of extending Prolog should be to increase its flexibility and efficiency without sacrificing its declarative nature.

As a historical example, consider the Prolog **not** operator, which is defined as unprovability. The **not** operator has a very simple computation rule (fail if succeed, succeed if fail) and was added as a nonlogical feature of Prolog. It was known to be nonlogical and programs that use it can have logically stange behavior. This occurs particularly if there are free variables in the scope of a **not** operator when it is evaluated. So programs containing **not** could not be counted on to have a declarative semantics. However, research of (Clark 1982, Jaffar et al. 1983) and others has shown that some programs that contain **not** can be given a declarative semantics. In order for **not** to have a logical semantics, the use of the operator must be constrained, so that it is not applied to terms with free variables. In addition, the underlying theory must be extended by predicate completion. In these circumstances, the Prolog **not** operator can be understood as logical **not**. Thus programs containing the **not** operator, appropriately constrained and interpreted, do have a declarative nature.

The **assert** operator in Prolog is notoriously nonlogical. **Assert** allows a programmer to change the set of clauses that make up the program or database being executed. As a Prolog operator, **assert** is often condemned and often used. It seems necessary for various reasons, including at least the following: (a) implementing a general database updating facility, (b) implementing complex AI systems including perhaps models of actors' knowledge and beliefs, and involving some kind of truth and/or consistency maintenance, and (c) making up for the lack of a suitable garbage collection facility in the Prolog system. The goal of this paper is to describe how a logical semantics can be provided for a restricted form of **assert**. Achieving this will allow programs which use assert for certain database manipulations to be understood declaratively.

In the next section, we discusses various aspects of the Prolog **assert** operator that make it nonlogical and thus difficult to understand in the same way that pure predicates are understood. Section 3 describes two ways, one characterized as syntactic and the other as semantic, in which modal logic might be used to provide a logical understanding of Prolog programs involving database updates. The approach we take in this paper is the semantic one. Section 4 introduces our new operator, **assume**, describes its operational characteristics, and compares it to Prolog's **assert**. Section 5 discusses some pragmatic issues involved with **assume**. Sections 6 and 7 provide a formal syntax and semantics for pure Prolog programs including **assumes**, understood as formulas in a modal logic. In Section 8 we give a resolution-based deduction method for our modal logic. Finally Section 9 describes a general extension to a standard Prolog interpreter, and discusses how it can be used to implement the modal deduction method.

2. THE EVILS OF ASSERT

We begin by exploring the ways in which the **assert** operator is nonlogical. That is, we show how the semantics of programs that include **assert** violate the normal logical semantics of pure Prolog queries. This can make such programs very hard to understand.

The first and most obvious difference between pure Prolog clauses and those that involve **assert** is the fact that, unlike pure clauses, the meaning of a clause involving **assert** depends very much on the order in which the literals appear in a clause. As a very simple example consider:

```
(1) :- assert(p(a)), p(a).
(2) :- p(a), assert(p(a)).
```

The meanings of (1) and (2) are clearly different. Query (1) adds the fact that **p** is true of **a** to the database and then asks if **p** is true of **a**. Clearly (1) always succeeds. Query (2) first asks if **p** is true of **a**. If it is not, then this query fails. Changing only the order of the literals changes drastically the meaning of the program. The ',' which for pure clauses is conjunction, is something else when the clause involves **assert**.

Another impurity of **assert** arises from the fact that it changes the global state even across backtracking. That is, an asserted clause stays in the database when the asserting operation is backtracked over. This introduces another kind of order dependence among clauses. As an example use of assert consider the following:

```
find(Z) :- .....
process :- find(X), assert(found(X)), fail.
process :- found(Y), playwith(Y).

:- process.
```

We assume that execution of **find** will bind the variable X. This use of assert then collects the values for which the predicate **find** is true and stores them in the database. Then the second clause of **process** retrieves them from the database and does some further processing. This construction releases the stack space used by **find** so that it is available for use by **playwith**. Another related reason for using the persistence of **assert** to compute values once and collect them in a table is to avoid recomputation costs in the case that the results will be used many times.

Another nonlogical aspect of queries including **assert** involves the meaning of variables. Consider:

```
(1) :- playwith(p(X)), q(X).
```

(2) :- assert(p(X)), q(X).

Assuming that **playwith** is a pure predicate, the variable X in query (1) is understood as being existentially quantified, and the two occurrences of the variable refer to the same object. In query (2), which has exactly the same structure, the two occurrences of the variable X mean very different things. The X in the scope of the **assert**, which is free when the assert is executed, is more related to a universal quantifier: the fact (for-all X p(X)) is added to the database. That X has nothing whatsoever to do with the X occurring in the next literal as an argument **q**. Indeed

(3) :- assert(p(Y)), q(X).

has semantics identical to that of (2). However, changing the first occurrence of X in query (1) to Y has a radical effect on its semantics. The point is that a variable in the scope of an **assert** plays a different role; it is not an object language variable.

The point of this list of aspects of **assert** is to show some of the obstacles to providing **assert** with a declarative semantics consistent with the rest of Prolog. In the following we propose another operator, called **assume**, that can take the place of some uses of **assert**, in particular those uses which implement simple relational database updates. We will be able to provide a logical, declarative semantics for **assume** that is consistent with pure Prolog. It will be instructive to see how the above problems are treated (or avoided) with **assume**.

3. LOGICAL THEORY OF UPDATES

The approach we take to providing a logical semantics for updates is to use a modal logic. We extend the first-order logic on which Prolog is based to a modal logic in which reference is made to multiple possible worlds. The **assume** operator will then be a modal operator that is given semantics as a relation on possible worlds, in a way originally proposed by Kripke. Execution of a Prolog program will then become a deduction in the modal logic.

Historically, modal logic was developed in an attempt to provide a formal semantics for the natural language concepts of necessity and possibility. There developed two approaches to providing a formal semantics for modal logical languages, known as the syntactic approach and the semantic approach. In the syntactic approach (Carnap 1947) an attempt was made to identify necessity with provability, a syntactic notion. The corresponding approach to providing a meaning for **assert** would be to view the entire Prolog database as a syntactic object: a set of axioms. Then **assert** is understood as changing the set of axioms. A query evaluation is understood as a derivation from an explicit (or implicit) set of axioms. So with this approach, executing a Prolog program is understood as proving a theorem, and programs have an explicit metalanguage component.

The second approach to providing meaning for classical modal logics is the semantic approach, due to Kripke (Kripke 1959). In this approach the definition of a model is extended so that a model includes a set of possible worlds, and an accessibility relation among them. A truth value is given to a modal formula with respect to a model and a possible world by referring to what worlds are accessible from the given world and the truth values of subformulas in those worlds. With the semantic approach to **assert** developed in this paper, programs with **assert** are defined as relations on possible databases. This means that such Prolog programs can be understood as moving the system from one database (world) to another. Dynamic Logic (Harel 1979) similarly uses a Kripke modal semantics to provide meanings for programming languages.

The syntactic approach to updates in Prolog has been explored by Kowalski and Bowen in (Bowen and Kowalski 1982). Carnap's syntactic approach to classical modal logic ran into certain technical problems (Montague 1963). Kowalski's and Bowen's approach retains the distinction between language and metalanguage and avoids those problems. This syntactic approach is very powerful in that it allows extremely general changes to the set of axioms. Thus it is an appropriate approach to the general problems of representing systems of knowledge and belief, that involve some aspect of truth maintenance. The power is not achieved without a cost, however. Problems involving truth maintenance and belief systems are notoriously intractable. For the particular problem of database updates, a syntactic approach was taken in (Fagin et al. 1983). That effort clearly shows how difficult it is to deal with the added complexity that comes from trying to allow absolutely any change to the set of axioms.

Our semantic approach to Prolog's **assert** is more limited than the syntactic. Since it views changes to the database as changes to the underlying worlds, we are essentially constrained to modify the database only by adding (or deleting) facts, as described below. Even though the semantic approach is more limited in scope, it is worth pursuing for several reasons. First, because it is more limited, it avoids some of the intractable problems raised in the syntactic approach. Second, in the area of traditional database theory, it is more appropriate to understand database updates as changes to the model rather than as changes to a syntactic entity: a set of axioms. Query processing is more naturally viewed as data retrieval from a model, rather than as theorem proving from a set

of axioms. Of course, it can also be understood as theorem proving (see (Nicolas and Gallaire 1979)). Indeed, this is the basis of the power of logic and logic programming as discussed above. But, in the context of traditional databases, it seems more natural to exploit the model theoretic, declarative, aspect. Third, there is really not a competition between the syntactic and semantic approaches; they are not mutually exclusive but are, in fact, complementary. A good understanding of the semantic approach may provide insight into how the syntactic approach should be developed.

4. ASSUME

We introduce the new **assume** operator by giving examples and explaining how it differs from, and is similar to, a restricted form of Prolog's **assert** operator.

Assume is a modal operator and as such applies to formulas. It has a different syntactic role than a predicate symbol does, so it also has a new syntax. For example:

:- assume(emp(john,25k)) @ (true).

Here **assume(emp(john,25k))** is a modal operator. It is applied (@ is the application symbol) to the pure Prolog conjunction **true**. Since **assume** is a modal operator and applies to formulas, we give it a special syntax that displays its modal role. This speaks to the nonlogical aspect of **assert** described above involving order dependencies of literals involving **assert**. As noted, with **assert**, the ',' is not conjunction; it is application. With **assume** application has its own symbol: '@'. Semantically distinct concepts should have a syntactically distinct forms of expression.

The effect of the above query is to add the fact **emp(john,25k)** to the database, i.e., the tuple <john,25k> is inserted into the relation emp. Then the query **true** is executed on the new database. Declaratively, this query is understood as saying that this formula is true if there is a database (world) which differs from the current one by having the fact **emp(john,25k)** true and **true** is true in that database. We could put any conjunctive query in place of the **true**, and it would be evaluated in the new database.

The **assume** operator is constrained to insert only facts, and not rules. Intuitively, this is because it is easy to determine of two worlds whether they differ from each other by a single fact; it is not so easy to determine whether two worlds differ from each other by a rule. This is the main constraint imposed by our semantic approach to updates. It may seem that this is a severe limitation, but note that traditional database systems allow only updates to base relations and, as we will see below, interesting problems do remain.

Unlike **assert**, facts added by **assume** are removed from the database on backtracking. The feature of **assert** of being persistent through backracking is logically somewhat problemmatical anyway, and it has been noted that it disconcerts many novice Prolog programmers. As an example of backtracking through **assume**, consider the following:

assume(emp(john,55k)) @ (avsal(X), X < 30k).

(Extra parentheses can be eliminated by assigning appropriate priorities to the infix symbols. We would give the ',' infix operator and the '@' infix operator the same priority and have them associate to the right.) This query adds the tuple <john,55k> to the database. Then it computes the average salary of all employees and checks whether that number is less than 30k. If not, it fails back through the **assume**, and that tuple is removed from the database. This might be an appropriate query if there is a policy that the average salary can never equal or exceed 30k. This query ensures that the tuple <john,55k> will not be added to the database if it would cause a violation of this constraint.

Consider the interpretation of variables that appear in **assumed** facts. **Assumed** facts are allowed to contain variables, but unlike in Prolog's **assert**, such variables have an existential interpretation. That is, when a fact containing a free variable is **assumed**, the new database (world) differs from the current database (world) by exactly one tuple. But exactly what tuple that is remains partially unspecified. If a variable in an **assumed** fact is bound to a constant value later in the computation, then that constant value replaces the variable in the **assumed** fact; the result is equivalent to having **assumed** the closed fact originally. For example:

(1) :- assume(emp(john,X)) @ X = 55k.

(2) :- assume(emp(john,55k)) @ true.

have identical effects. (1) is understood as claiming the existence of a world that differs from the current one in having an **emp** tuple with first component **john** and some unknown second component, and that the unknown second component is equal to **55k** in that world. This is equivalent to (2) which claims the existence of a world that differs from the current one in having an **emp** tuple with first component **john** and second component **55k**, and **true** is true in that world. We emphasize the existential nature of the variable; a world claimed to exist by an **assume** operator differs from the current world by at most one tuple. There is no way to assume a fact with a universally bound variable. Notice that this treatment of variables in **assumed** facts is intuitively consistent

with the treatment of variables in pure Prolog programs. For example, the literal **X=55k** could be put before the **assume(emp(john,X))** in the above example and the semantics would not be changed. Pure Prolog programmers' intuitions are still correct.

As as slightly more complicated example, consider the query:

```
:- Ssn=111,
   (assume(empdemo(Ssn,john,Addr)) @
      (Addr='5 Shady Lane',
       assume(emptax(Ssn,Sal,2)) @ (Sal=55k))).
```

We might issue a query such as this to update the database to reflect the addition of a newly hired employee. This query is understood logically as claiming that there is a world (call it b) accessible from the current one (call it a) by assuming a tuple in **empdemo**, and the embedded modal formula is true there. I.e., from world b there is a world (call it c) accessible by assuming a tuple in **emptax**. (The '=' are given here in a perhaps unlikely arrangement just to show variety.) Notice that we have allowed the argument of the first modal operator **assume(empdemo(Ssn,john,Addr))** to be a modal conjunctive formula:

```
(Addr='5 Shady Lane',assume(emptax(Ssn,Sal,2)) @
      Sal=55k)
```

It is a conjunction of literals, one of which is an **assume**. We will constrain any conjunction of literals to contain at most one modal subformula. This is to ensure that the resulting database is uniquely determined. For the query above, the resulting database is clearly the one with the two new tuples in it. Consider what would happen were we to allow a query such as:

```
:- (assume(empdemo(111,john,'5 Shady Lane')) @
      true) ,
   (assume(emptax(Ssn,55k,2)) @ true).
```

This query includes two parallel modal subformulas: it is true if there exists a database different from the current one by including an **empdemo** tuple and if there is another world different from the current one by including an **emptax** tuple. This query claims the existence of two different worlds, and it is not clear in which processing should continue. While it is tempting to try use such formulas to develop a theory of alternative databases, this turns out not to be the way to include such nondeterminism. While such 'branching' queries might be acommodated in the main query, we want to allow modal operators to be defined by rules. Therefore, we exclude such formulas and permit only a linear nesting of modal operators.

To this point we have only considered uses of the **assume** operator in the main query. The use of **assume** would be extremely limited were we not able to define rules using **assume** operators. Such rules are required in order to construct complex operators that act on and change multiple relations in the database. Note that a rule whose antecedent contains an **assume** operator defines another modal operator, not a predicate. The meaning of a symbol appearing in the consequent of such a rule is a relation on possible worlds and is a user-defined modal operator. Thus the syntactic (and semantic) role of such a symbol is similar to that of **assume** itself; it is applied to a conjunction using the '@'. Also, since rules that define modal operators are semantically distinct from normal Prolog rules, which define pure predicates, there is a different syntax for them. Instead of the ':-' sign, a '←' sign is used for defining modal operators.

As a simple example consider:

```
hire(Empname,Ssn,Addr,Sal,Deds) ←
      not(empdemo(Ssn,E,A)),
      (assume(empdemo(Ssn,Empname,Addr)) @
        (assume(emptax(Ssn,Sal,Deds)) @ true)).
```

This modal rule defines **hire** as a modal operator. Operationally **hire** can be understood as an operator that takes a sequence of values describing a new employee, and checks to see whether the employee is already in the database. If so, it fails. Otherwise it inserts the appropriate tuples in the employee demographic relation and the employee tax relation. Declaratively it can be understood as defining **hire** to be a (parameterized) binary relation on worlds. This relation is defined by the semantics of the modal formula that makes up the antecedent of the rule. That formula says that the current world, say w1, does not have a tuple in **emp** with the given identifying number (Ssn), and from w1 there is a world, say w2, accessible through an assume, and from w2, there is another world, say w3, accessible through another assume. So the (linear) modal formula can be given a modal semantics as the relation on worlds defined as all possible such <w1,w3> pairs. The modal rule states that the meaning of the modal operator **hire** is a superset of this relation. So given this rule, the following query:

```
:- hire(john,111,'5 Shady Lane',55k,2) @ true.
```

adds two new tuples to the database, if John really is a new employee.

It is sometimes desirable to have nondeterministic updates. This is achieved by using more than one rule to define a modal operator. Consider the situation in which students are enrolled into a course which has several sections, and each section has an enrollment cap.

```
enrl(Nm,s1) ← assume(s1(Nm))@size(s1,N),N<30.
enrl(Nm,s2) ← assume(s2(Nm))@size(s2,N),N<25.
enrl(Nm,s3) ← assume(s3(Nm))@size(s3,N),N<28.
```

This is a very simple approach which defines a modal operator **enrl** to enroll a student in a section that has openings. Standard Prolog backtracking first tries to enroll a student in section s1, succeeding if the enrollment cap is not exceeded; if s1 is full, it tries s2, and then finally s3. This algorithm makes the change to the relation and then checks to see if the result is consistent. An optimizer might transform this query into one which does the check first. However, there might well be cases in which it is simpler to make the change and then do a complex check for consistency.

One might easily add a operator that calls this one that takes a list of preferences and tries them in the indicated order:

```
enrollpref(Name,Prefs) ←
    member(Sec,Prefs),enrl(Name,Sec)@true.
```

5. PRAGMATICS

Recall that all **assumed** tuples are removed from the database on backtracking. This raises the question of how any database change can become permanent. Many Prolog systems, given a query, find a successful path through it and display the resulting variable bindings for the user. The user then has the option of either accepting the answer and telling the system not to search for any other answers (entering a cut), or telling the system to search for more answers (entering a fail). Entering a cut discards backtracking points and can be thought of as making it impossible for the system to backtrack. This is precisely what is needed in order to make changes database permanent: the elimination of the possibility of backtracking. In this respect, a Prolog cut corresponds to the database notion of 'commit'; it is the point at which the changes made to the database become permanent because it can no longer be removed by backtracking. Thus an **assumed** tuple can be considered as being permanently entered in the database when the user responds with a 'cut' to the answer produced by the Prolog system.

Consider now the situation in which an **assumed** tuple still contains free variables at the time of the commit. This would cause there to be existential variables in the permanent Prolog database. This, in principle, causes no logical problems. Such variables must be treated similarly to those in **assumed** facts before a commit. The Prolog system must be extended to allow such free variables to occur on permanently stored databases. This essentially involves an implementation of Prolog's runtime management of variables and their bindings on the permanent database.

These existential variables in facts in the database act as a kind of 'null values'. They stand for values that exist, but are at the current time unknown. They can be filled in by later queries, which bind those variables. Consider the example:

```
:- hire(john,111,'5 Shady Lane',Sal,2)@(true,!).
```

This adds a tuple with an undefined value for John's salary to the emptax relation. The cut (!) makes this a permanent change to the database. Then perhaps the personnel department wants to fill in John's salary with 55k. They can enter the following query:

```
:- emptax(111,Sal,Ded), Sal=55k, !.
```

This will match John's tuple and will bind his salary field to 55k. The cut makes the change permanent. Note, however, that if this is allowed, then a pure logical query, as this one is, can change the database. Clearly this is not always what is desired. Were this allowed the system would let a null value satisfy any query concerning it. Logically, there is no problem; logically the system is saying that the above query is true *if John's salary is 55k*. This is a conditional response to the query as we will see when we look in detail at the formal semantics for **assume**. Pragmatically, we might constrain the system so that only authorized users could cause changes to become permanent. Other users would be told that they are trying to bind the value of a null value in the database and would not be allowed to change it. Note that it is also possible for a query to equate two free variables in the database (cf. the chase, as described e.g. in (Maier 1983)).

6. FORMAL MODAL SYNTAX

We give here a formal definition of the syntax of Prolog programs including **assume** as a modal operator.

Symbols: There are countably many variables $x_0, x_1, \ldots$. There is a set of function symbols of various arities, a set of predicate symbols of various arities, and a set of operator symbols of various arities. All these sets are disjoint. The zero-ary function symbols are called constants.

Terms: A variable is a term. A constant is a term. If f is an n-ary function symbol ($n \geq 1$) and $t_1, t_2, \ldots, t_n$ are terms, then $f(t_1, t_2, \ldots, t_n)$ is a term.

Atomic Formulas: If p is an n-ary predicate symbol and $t_1, t_2, \ldots, t_n$ are terms, then $p(t_1, t_2, \ldots, t_n)$ is an atomic formula. If n is zero, the parentheses are deleted.

Conjunctive Formulas: If $p_1, p_2, \ldots, p_n$ ($n \geq 1$) are atomic formulas, then $p_1, p_2, \ldots, p_n$ is a conjunctive formula. Note, an atomic formula is a conjunctive formula.

Modal Formulas: If o is an n-ary operator symbol, $t_1,t_2, \ldots, t_n$ (n $\geq$ 1) are terms, and s is a conjunctive formula or a modal conjunctive formula, then $o(t_1,t_2, \ldots, t_n)@(s)$ is a modal formula. For zero-ary o, $o@(s)$ is a modal formula.

Modal Conjunctive Formulas: A modal formula is a modal conjunctive formula. If p is a conjunctive formula and q is a modal formula, then p,q is a modal conjunctive formula. Note, a modal conjunctive formula is a conjunction of formulas exactly one of which is a modal formula.

Clauses: If p is an atomic formula then $p.$ is a clause. If p is an atomic formula and r is a conjunctive formula, then $p:-r.$ is a clause.

Modal Clauses: If o is an n-ary operator symbol, $t_1,t_2, \ldots, t_n$ are terms, and u is a conjunctive modal formula, then $o(t_1,t_2, \ldots, t_n)\leftarrow u.$ is a modal clause. If n is zero, the parentheses are deleted.

The examples given in the earlier sections exhibit this syntax. The one exception is the way the **assume** was written. We used a shorthand above. We actually need a different **assume** operator for each relation to which tuples are to be added. For example, instead of writing

assume(emptax(111,55k,2)) @ true

we would write

assume_emptax(111,55k,2) @ true

where **assume_emptax** is a 3-ary modal operator. Operationally one may think of this as forcing the obvious requirement that the predicate name be known before an insert can be performed. If the earlier syntax is preferred by a Prolog programmer, the definitions similar to those suggested in (Warren 1982) could be used.

7. FORMAL MODAL SEMANTICS

A modal structure S for Modal Prolog is a 3-tuple $<D,W,F>$, where D is a set of individuals, W is a set of worlds, and F is an interpretation function as follows:

(1) For an n-ary function symbol f, $F(f):D^n \rightarrow D$, i.e., $F(f)$ is an n-ary function on D.

(2) For an n-ary predicate symbol p, $F(p) \subset W \times D^n$, i.e., $F(p)$ determines, for each w, an n-ary predicate on D.

(3) For an n-ary operator symbol o, $F(o) \subset W \times W \times D^n$, i.e., F(o) determines, for each n-tuple of individuals from D, a binary relation on W.

Given a structure S, a variable assignment v is a function from variables into D. We give here the definitions for the interpretation (I_v) of a term in a structure given a variable assignment v, the truth ($I_{v,w}$) of a formula in a structure given a world and a variable assignment, and the truth (I) of a clause. The structure S is asuumed to be understood.

Terms: If x_n is a variable, then $I_v(x_n)$ is $v(x_n)$. $I_v(f(t_1,t_2, \ldots, t_n))$ is $F(f)(I_v(t_1),I_v(t_2),\ldots,I_v(t_n))$. For constant c, $I_v(c)$ is $F(c)$.

Atomic Formulas: $I_{v,w}(p(t_1,t_2, \ldots, t_n))$ is true if
$<w,I_v(t_1),I_v(t_2),\ldots,I_v(t_n)> \in F(p)$.

Conjunctive Formulas: $I_{v,w}(p_1,p_2, \ldots, p_n)$ is true if $I_{v,w}(p_i)$ is true for $i=1,\ldots,n$.

Clauses: $I(p.)$ is true if for every variable assignment v and every world w, $I_{v,w}(p)$ is true. $I(p:-r.)$ is true if for every variable assignment v and every world w, if $I_{v,w}(r)$ is true, then $I_{v,w}(p)$ is also true.

Note that the under these definitions the interpretation of a term does not depend on the world in which it is evaluated. This is because a function symbol has the same interpretation in every possible world of a structure. Note also that a clause does not depend on the world in which it is interpreted. It holds in a structure if it holds in every world in the structure.

We now give the definitions for the interpretations of formulas containing modal operators. We define two interpretations for both modal formulas and modal conjunctive formulas. For a modal formula u, $I_{v,w}(u)$ is a truth value, and $I_v'(u)$ is a binary relation on W.

Modal Formulas: $I_{v,w}(o(t_1,t_2, \ldots, t_n)@(s))$ is true if there is a w' such that $<w,w'> \in I_v'(o(t_1,t_2, \ldots, t_n)@(s))$
For s a conjunctive formula, $<w,w'> \in I_v'(o(t_1,t_2,\ldots,t_n)@(s))$ iff
$<w,w',I_v(t_1),I_v(t_2),\ldots,I_v(t_n)> \in F(o)$, and $I_{v,w'}(s)$ is true.
For s a modal conjunctive formula, $<w,w'> \in I_v'(o(t_1,t_{2,\ldots,t_n})@(s))$ iff there is a w'' such that $<w,w'',I_v(t_1),I_v(t_2),\ldots,I_v(t_n)> \in F(o)$, and $<w'',w'> \in I_v'(s)$.

Modal Conjunctive Formulas: $I_{v,w}(p,q)$ is true if $I_{v,w}(p)$ is true and $I_{v,w}(q)$ is true. $<w,w'> \in I_v'(p,q)$ if $I_{v,w}(p)$ is true and $<w,w'> \in I_v'(q)$.

Modal Clauses: $I(o(t_1,t_2, \ldots, t_n)\leftarrow u.)$ is true if for every variable assignment v, $I,v(u) \subset I,v(o(t_1,t_2, \ldots, t_n)@(true))$

A model for a set of clauses is a modal structure in which for each clause r, $I(r)$ is true. Since the only way to constrain modal operators is by using modal clauses, there must be some predefined operators in order to be able to define any non-

trivial operators at all. Thus we introduce the following logical modal operators. Given a modal structure $<D,W,F>$, for each n-ary predicate p, there is an n-ary operator *assume_p*.

For $a_1,a_2,\ldots,a_n \in D, <w_1,w_2,a_1,a_2,\ldots,a_n> \in F(assume\_p)$ iff

(1) for every predicate q other than p,
  for all $<b_1,b_2,\ldots,b_n> \in D^n$,
    $<w_1,b_1,b_2,\ldots,b_n> \in F(q)$ iff
    $<w_2,b_1,b_2,\ldots,b_n> \in F(q)$.

(2) for all $<b_1,b_2,\ldots,b_n> \in D^n$ such that
  $<b_1,b_2,\ldots,b_n> \neq <a_1,a_2,\ldots,a_n>$,
    $<w_1,b_1,b_2,\ldots,b_n> \in F(p)$ iff
    $<w_2,b_1,b_2,\ldots,b_n> \in F(p)$.

(3) $<w_2,a_1,a_2,\ldots,a_n> \in F(p)$.

This definition just formalizes our intuitions that two worlds are **assume_p(t)** related if they agree on all predicates other than **p**, agree on **p** for all tuples other than **t**, and **p** in the second world contains **t**.

8. FORMAL DEDUCTION

We now briefly discuss a resolution-based deduction method for this modal logic. The method takes a set of clauses, a set of modal clauses, and a modal conjunctive formula, called the goal. It determines whether there exists a modal structure that satisfies the clauses, the modal clauses, and the negation of the goal. (The negation of a formula is defined in the standard way: the negation of a formula is true in a world of a structure iff the formula is false in that world.) Thus it can be used as a refutation proof procedure.

The method described here is a simple extension of resolution and looks very much like the SL-resolution on which Prolog is based. This is intentional; the aim is to provide a formal logical semantics for the operation of a Prolog program that includes a limited form of **assert**.

The method is nondeterministic. It works on the goal formula and maintains a set of assumed literals. The deduction succeeds when the goal has been reduced to empty:

(1) Case 1: The goal formula is a conjunctive modal formula whose first symbol is a predicate symbol; then nondeterministically choose (a) or (b): (a) Choose any clause (changing bound variables if necessary) whose head unifies with that first literal. Add the antecedent of the unifying clause as a prefix to the goal, and apply the unifying substitution to the entire goal, and the set of assumed literals. (b) Choose any assumed fact (without changing bound variables) which unifies with that first literal of the goal. Remove the matching literal from the goal formula. Apply the unifying substitution to the entire goal formula and to the set of assumed literals.

(2) Case 2: The goal formula is a modal formula; then choose (a) or (b): (a) Choose a modal clause that unifies (changing bound variables if necessary) with the modal operator (and its arguments) that is the main operator of the goal. Construct the new goal by taking the antecedent of the modal clause, and changing the inner-most conjunctive formula into a modal conjunctive formula by conjoining the formula to which the matched operator was applied in the goal. Apply the unifying substitution to the resulting goal and the set of assumed literals. (b) If the main modal operator of goal formula is a **assume-p(t)** then add **p(t)** to the set of assumed literals.

As a very simple example consider an execution of a query in a database including a modal clause:

```
hire(Empname,Ssn,Addr,Sal,Deds) ←
      assume_empdemo(Ssn,Empname,Addr)@
        (assume_emptax(Ssn,Sal,Deds)@Sal<50k).

:- hire(john,111,'5 Shady Lane',30k,Deds)@
      (Deds = 2).
```

We trace a deduction of the null clause:

```
Goal: hire(john,111,'5 Shady Lane',30k,Deds)@
      (Deds = 2)
Assumed Lits:      none

Goal: assume_empdemo(111,john,'5 Shady Lane')@
        (assume_emptax(111,30k,Deds)@
          (30k<50k,Deds = 2))
Assumed Lits:      none

Goal: assume_emptax(111,30k,Deds)@
        (30k<50k,Deds = 2)
Assumed: empdemo(111,john,'5 Shady Lane')

Goal: 30k < 50k, Deds = 2
Assumed: empdemo(111,john,'5 Shady Lane')
        emptax(111,30k,Deds)

Goal: Deds = 2
Assumed: empdemo(111,john,'5 Shady Lane')
        emptax(111,30k,Deds)

Goal:
Assumed: empdemo(111,john,'5 Shady Lane')
        emptax(111,30k,2)
```

It is not difficult to show that this refutation procedure is sound. Let L' and G' be the new assumed literals and the new goal derived from L and G by one step of this modal resolution. Let A be the set of clauses. The main step is to show that if there exist a structure S, a world w, and a variable assignment v such that (a) for each r in A, $I(r)$ is true, (b) for each a in L, $I_{v,w}(a)$ is true, and (c) $I_{v,w}(\neg G)$ is true, then there exist w' and v'

of S such that for each a' in L', $I_{g',w'}(a')$ is true, and $I_{g',w'}(\neg G')$ is true. (For this we assume the initial set of assumed literal contains just the literal **true**.) From this it follows that if there exists a structure, world and variable assignment satisfying the initial axioms and the negation of the goal, then there exists a structure, world and variable assignment satisfying the negation of any derived goal. If we can derive the empty clause as a goal, then there must be no structure and world satisfying the axioms and the negation of the initial goal. Thus every structure satisfying the axioms satisfies the initial goal and we have a refutation proof procedure.

9. CONDITIONAL PROOFS

In this section we explore how a standard Prolog system can be used to execute the modal deduction method described previously. We first give a simple generalization of the standard Prolog execution algorithm. This generalization allows Prolog to construct conditional proofs and has has been proposed by several researchers in various guises for the solution of various problems. We briefly point out these uses and suggest how the same generalization can be used to implement our modal deduction method.

The following minor modification of Prolog's standard resolution algorithm allows Prolog to construct conditional proofs. Normally Prolog maintains a list of goals yet to be satisfied. We add another list of literals for Prolog to maintain, called the list of assumptions. We extend Prolog's evaluation strategy in two simple ways:

(1) At various points (to be specified by a control strategy) instead of resolving the next subgoal against the heads of the clauses, Prolog may instead simply move the subgoal to the list of assumptions. The intuition is that this will be an assumption; the following proof is valid if indeed this subgoal is true. Also Prolog at any point, instead of choosing the next subgoal to resolve on, may choose a literal from the list of assumptions to prefix to the list of remaining goals, and thus use it as the next subgoal.

(2) When trying to reduce a subgoal, in addition to trying to unify all heads of clauses, the system also tries to unify with literals on the list of assumptions. If it succeeds, the subgoal is satisfied, and the unifying substitution is applied to the assumptions and to the remaining goal list. The intuition is that this is an assumption that has been made in the course of the proof and we can use the assumption again.

Of course, any proposal for using this generalization of Prolog's control strategy must fully specify when subgoals are moved to the list of assumptions and when they are moved back.

This can be understood as an implementation of a resolution method that has three main sets of literals: the clauses C, the assumptions list A, and the goal list G. The assumptions list can be thought of as a list of literals that came to the head of the goal list but instead of resolving them away, they have been delayed. So logically they are still literals on the goal list. Normally we think of horn clause resolution as manipulating a set of clauses C,¬G where C is the set of horn clauses with one positive literal, and ¬G is the goal clause consisting of all negative literals. This new resolution modified to include assumptions can be thought of as manipulating a set of clauses C, ¬A or ¬G, where ¬A or ¬G is a single horn clause consisting of the assumptions and the remaining goals. The "point of activity" of the linear resolution theorem prover is at the first literal of ¬G.

There have been several suggestions of ways to use a Prolog system modified to maintain a set of assumptions. If these assumptions are evaluated later in the execution of the Prolog program, this this is a way of changing Prolog's strict left-to-right, top-down ordering of evaluation of goals. The major issue is what goals have their evaluation delayed and why. Examples of such proposals include the following. (1) (Kornfeld 1983) proposes an omega operator for delaying the evaluation of certain unifications until enough variables have been bound. He does not use one assumption list but associates his delayed subgoals with variables that they involve. From the point of view proposed here, this could be thought of as a way of indexing the literals on the assumption list. (2) (Vassiliou et al. 1983) and (Yokota et al. 1984) suggest using an assumptions list to delay the evaluation of literals that would involve retrieval from large database relations stored on disk. They propose collecting together all the assumptions lists from every successful path through the Prolog program, and giving that union as a query to a relational database query processor. That query can then be optimized and evaluated, producing the answer to the original Prolog query. (3) (Hsiang and Srivas 1984) propose using an assumptions list to delay the evaluation of certain recursive goals, allowing them to produce a set of implications that need to be proved to establish the original Prolog query. They then use other methods to prove these implications and thus are able to use Prolog to help prove inductive properties of data structures.

The basic function the assumptions list serves is to allow Prolog to be a problem reducer, instead of a problem solver. Extended to include an assumptions list, Prolog will produce, as an answer to a query, a list of variable bindings *and* an assumptions list. This can be understood as being a conditional answer: the variable values are an

answer to the query *if* all of the assumptions are true. In all cases above, the system then continues on to try to prove the conditions so that the final answer produced by the system has no hedges. Note that another type of theorem prover might be invoked to attempt to prove the assumptions list.

Consider now how the assumptions list is used in our case, for the implementation of database inserts. We can view an **assume** operation as being an explicit direction to the Prolog system to continue on the assumption that the given fact is true, i.e. to move the subgoal to the assumptions list. The commit (cut) tells the system to go ahead and change the database so that the assumptions are made to be true. Thus after the commit is done, the answer is no longer conditional; the database has been changed to include what had previously been conditions; we turn our conditional answer into an absolute answer by making the conditions true "by fiat".

10. SUMMARY AND FUTURE WORK

We have proposed the addition of a new operator, **assume**, to Prolog to take the place of uses of **assert** that maintain relational databases. We have argued that programs with **assume** retain a declarative semantics, and we provided a modal logic to define formally that semantics. We then gave a simple extension to Prolog's SL resolution proof method and showed how it could be understood as a question-answering theorem proving method for the modal logic. We saw how the method of conditional proofs for Prolog, which is useful in many other contexts, can be used to implement the modal refutation procedure.

Much remains to be explored. It is important to show some sort of completeness property for the modal resolution proof method. Also with this formal semantic notion of the semantics of updates, we can now explore a formal theory of optimization of database queries with updates. We also wish to understand in more detail the use of cut as commit, and how it can be more generally applied. This is related to the concepts of nested transactions and nested commit (Moss 1981).

REFERENCES

Bowen, K. A., and Kowalski R. A., Amalgamating Language and Metalanguage in Logic Programming, in *Logic Programming,* K. L. Clark and S. -A. Taernlund, (eds.), Academic Press, New York, NY, 1982, 153-172.

Carnap R., *Meaning and Necessity,* University of Chicago Press, Chicago, 1947.

Clark K. L., Negation as Failure, in *Logic and Databases* J. Minker and H. Gallaire (eds.), Academic Press, New York, NY, 1982, 293-324.

Fagin R., Ullman J. D., and Vardi M., On the Semantics of Updates in Databases, *Proc. Principles of Database Systems,* Atlanta, March 1983, 352-365.

Harel D., *First-Order Dynamic Logic,* Springer-Verlag, Berlin-Heidelberg-New York, 1979.

Hsiang J., and Srivas M. K., On Proving First-Order Inductive Properties in Horn Clauses, Technical Report, SUNY Stony Brook, 1984.

Jaffar, J., Lassez, J. L., and Lloyd, J., Completeness of the Negation as Failure Rule, *Proceedings of IJCAI,* August 1983, 500-506.

Kornfeld, W. A., Equality for Prolog, *Proceedings of IJCAI,* August 1983, 514-519.

Kripke S., A Completeness Theorem in Modal Logic, *Journal of Symbolic Logic,* 24, (1959), 1-14.

Maier, D., *The Theory of Relational Databases,* Computer Science Press, 1983.

Montague, R., Syntactical Treatments of Modality, with Corollaries on Reflexion Principles and Finite Axiomatizability, in *Formal Philosophy: Selected Papers of Richard Montague,* 1963, 286-302.

Moss, J., Nested Transactions: An Approach to Reliable Distributed Computing, MIT Technical Report 260, 1981.

Nicolas, J. M., and Gallaire H., Data Base: Theory vs. Interpretation, in *Logic and Data Bases,* H. Gallaire and J. Minker, (eds.), 1979.

Vassiliou, Y., Clifford, J., and Jarke, M., Access to Specific Declarative Knowledge by Expert Systems: The Impact of Logic Programming, Technical Report, NYU School of Business, 1983.

Warren, D. H. D., Higher-order Extensions to PROLOG: Are They Needed?, *Machine Intelligence 10,* 1982, 441-454.

Yokota, H., Kunifuji, S., Kakuta, T., Miyazaki, N., Shibayama, S., and Murakami K., An Enhanced Inference Mechanism for Generating Relational Algebra Queries, *Proceedings of the Third Symposium on Principles of Database Systems,* April 1984, 229-238.