NEGATION IN LOGIC PROGRAMMING

KENNETH KUNEN*

▷ We define a semantics for negation as failure in logic programming. Our semantics may be viewed as a cross between the approaches of Clark [5] and Fitting [7]. As does [7], our semantics corresponds well with real PROLOG in the standard examples used in the literature to illustrate problems with [5]. Also, PROLOG and the common variants of it are sound but not complete for our semantics. Unlike [7], our semantics is constructive, in that the set of supported queries is recursively enumerable. Thus, a complete interpreter exists in theory, although we point out that there are serious difficulties in building one that works well in practice. ◁

1. INTRODUCTION

The question of negation in logic programming has been frequently discussed in the literature. The goal of these investigations has been to find a declarative semantics for negation which corresponds to the way PROLOG-style languages actually treat negation. We describe such a semantics here. The outcome of our approach is a semantics which is a cross between the completed database obtained by Clark [5] and the 3-valued logic approach advanced by Fitting [7]. Our semantics is more restrictive than either of these approaches, in the sense that any query which follows from the database under our semantics also follows under both [5] and [7]; but not conversely, as we will show by some examples.

Although our semantics is more restrictive, it is still true that any PROLOG-style interpreter will be *sound* under our semantics, provided that we remedy two features of standard PROLOG implementations which are commonly agreed to be in error. One is that we assume that unification is always done with an occurs check. The other is in the treatment of a negated nonground literal; we discuss this in greater detail in Section 3.

*Research supported by NSF Grant DMS-8501521.
Received April 1986; accepted December 1986.
Address correspondence to Professor Kenneth Kunen, Computer Science Department, University of Wisconsin, Madison, WI 53706.

The question of *completeness* is more difficult. In the ground (propositional) case, we describe an abstract interpreter which is both sound and complete for our semantics; this interpreter is essentially standard PROLOG, but with the user's ordering of the clauses and of the literals within each clause replaced by a nondeterministic (or a parallel) execution. The fact that our semantics corresponds exactly with such an abstract PROLOG, whereas [5] does not, could be taken as an argument in favor of our approach. For ground databases, our semantics is identical with that of [7]. In the nonground case, although there exists a complete evaluation procedure in *theory* (i.e., unlike in [7], the set of answers to queries is recursively enumerable), we do not believe that any such procedure is practical to implement.

The basic idea behind our approach, as in all others, is "negation as failure". *Roughly*, this means that if the interpreter fails to verify α, then $\neg\alpha$ is taken as verified. This is justified for two reasons. First, it corresponds to the way PROLOG works. Second, it is often more convenient for database applications than is pure resolution logic, since the database need only contain positive relational information, and need not include the negations of all false relations. The problems begin in trying to find a coherent formal semantics behind this rough idea.

2. SYNTAX

We use standard PROLOG-style syntax. Variables begin with capital letters. Identifiers beginning with lowercase letters can be constants, function symbols, or predicate symbols. Terms are constructed from variables, constants, and function symbols. Atomic formulas (or positive literals) are constructed from predicate symbols and terms. Arbitrary predicate logic formulas are built up from atomic formulas using propositional connectives and quantifiers.

We wish to think of PROLOG clauses as embedded syntactically in ordinary predicate logic, although the semantics will be nonstandard. To this end, we think of "not" and "$\neg$" as synonymous, and "," and "$\wedge$" as synonymous, and we think of "$\phi \rightarrow \psi$" as synonymous with "$\psi :- \phi$". For example, the PROLOG clause "$\alpha :- \beta, \gamma$" is equivalent with the predicate-logic formula "$(\beta \wedge \gamma) \rightarrow \alpha$". So that we don't have to make special cases for clauses with empty body, we think of our logic as including the atom *true*; thus, the clause "α." can be written as "$\alpha :-$ true".

Program clauses (which we expect to find in the database) are expressions of the form

$$\alpha :- \lambda_1, \ldots, \lambda_n.$$

In the semantics, these clauses are considered to be universally quantified. *Query clauses* are expressions of the form

$$\lambda_1, \ldots, \lambda_n.$$

They are thought of as being existentially quantified. In all cases, the head of a program clause, α, is an atomic formula. We will consider three possibilities for what the λ_i in the body of a program clause or in a query clause can be. In *pure Horn clause* logic, they must also be atomic formulas. In *generalized clause* logic, they can be arbitrary literals (atomic formulas or their negations). In *extended clause* logic, they can be arbitrary formulas of predicate logic. The terminology for these

last two notions is borrowed from [17]. It is the generalized program clauses which correspond most closely to standard PROLOG.

The semantics for pure Horn clauses is well known and noncontroversial (see [16]); the difficulties lie with the passage to generalized clauses and the interpretation of the negations in the body of these clauses. Once this difficulty is conquered, there seems to be no problem dealing with extended clauses. [17] gives a translation from extended program clauses into generalized program clauses and shows this translation to be valid for Clark's semantics; it is also valid for Fitting's semantics [7] and our semantics (Theorem 6.7). Thus, although we shall define our semantics for extended program clauses, our theorems and examples will be concentrated mainly on generalized program clauses.

The fact that we are making our basic definitions for extended program clauses will allow us a syntactical simplification. That is, we may replace each body of a program clauses by its conjunction (or *true* if the body is empty), so that every extended program clause is simply of the form

$$\alpha :- \phi.,$$

where α is an atomic formula and ϕ is any predicate logic formula.

If ϕ is any formula, we let $\forall\phi$ be the universal closure of ϕ (formed by universally quantifying all free variables), and $\exists\phi$ the existential closure.

For convenience, we follow the usual convention in mathematical logic and assume that the same identifier cannot be used in two different contexts. This departs from standard PROLOG, where for example,

$$p(X, p(Y)) :- p.$$

is perfectly legal. The PROLOG interpreter views the three occurrences of p as completely unrelated symbols: a 2-place predicate, a 1-place function, and a 0-place predicate (proposition letter). Our convention obviously does not affect the basic theory.

3. IDEAL LOGIC PROGRAMMING

We consider the abstract model of a PROLOG-style interpreter, where there is a database, DB, on line, consisting of a finite set of program clauses, and the user addresses query clauses, ϕ, to the database. Recalling that ϕ is thought of as being existentially quantified ($\exists\phi$), our ideal interpreter should return one of three kinds of answers. The first is "no" ($\exists\phi$ is not a consequence of DB). The second is one or more substitutions, σ (i.e., $\forall\phi\sigma$ is a consequence of DB); if ϕ is ground, so that σ is irrelevant, the answer can just be "yes". In these two cases, we call the "no" or the "σ" a *solution* to the query. The third answer possible is "unclear". By "unclear", we mean either that the interpreter in fact fails to halt, or that it halts and prints an error message rather than a solution.

The goal of logic programming is summarized by Kowalksi's "Algorithm = Logic + Control" [12]. We take this to mean that the analysis of a PROLOG-style language can be broken up into a *logic component* and a *control component*.

The *logic component* should give us a declarative semantics of the language which tells us whether or not $\forall\phi\sigma$ indeed follows semantically from DB. The exact nature of this declarative semantics is a matter of some controversy which this paper partly

addresses. The problems with negation and other PROLOG constructs make it clear that the semantics cannot be precisely the standard semantics of first-order predicate logic, but it is important that it be *declarative*—that is, phrased in terms of logical notions such as truth, model, and satisfaction. The user should be able to understand the semantics just by "logic", not by a detailed understanding of the implementation of the interpreter. It is really this key fact which separates logic programming from more conventional procedural forms of programming.

The *control component* consists of guides to the PROLOG interpreter to help it decide whether or not a query is a logical consequences of DB. In PROLOG, the primary control mechanism is the ordering given to the clauses in the database and to the literals in each clause; many more sophisticated mechanisms have been advanced in the literature. Since the control should not affect the semantics, it is not the subject of this paper. We remark only that many common PROLOG features (such as cut), which are often referred to as "control" in the literature, *do* affect the semantics and in fact render the interpreter unsound.

The basic requirement of any PROLOG interpreter should be *soundness*; that is, if an answer σ is returned for a query ϕ, then $\forall\phi\sigma$ does in fact follow semantically from DB, while if "no" is returned, then $\exists\phi$ does not follow. We view the machine's answer of "unclear" as a lack of commitment as to whether $\exists\phi$ does or does not follow, so that any interpreter which always returns "unclear" is trivially sound (and useless).

We remark now on the problem of "not" applied to literals with variables. This is illustrated by the following example (from [16, p. 82]):

q :– not$(r(X))$, $p(X)$.

$p(a)$.

$r(b)$.

Conventional PROLOG will return a "no" to the query " :– q?", whereas it would return a "yes" if the ordering on the literals in the body of the first clause is reversed. This behavior must be unsound under any semantics in which $\wedge$ is commutative. Here, most approaches to PROLOG semantics agree that the correct answer is "yes". The problem is that the correct semantics views the subgoal "not$(r(X))$" as with a $\exists$ understood (as in all goals); here it is $\exists X\neg r(X)$; while the interpreter runs as if it were testing for $\neg\exists X r(X)$ (and fails when some $X = b$ satisfying r is found). There are various possibilities for what a sound PROLOG interpreter should do if it is asked to evaluate "not $r(X)$". The simplest answer, given by Clark [5], is that it should simply return "unclear". More complicated answers, such as those used in the systems IC-PROLOG [6] and MU-PROLOG [20, 21], will return solutions for more general classes of queries, but these systems are still sound for the semantics described in this paper (see Section 9). In any case, our soundness results apply only to systems which correctly understand a goal of $\neg\phi$ to mean $\exists(\neg\phi)$ and not $\neg(\exists\phi)$.

4. COMPARISON WITH OTHER APPROACHES

Many of the key ideas of our semantics occur elsewhere in the literature, but not with the same outcome.

One idea, due to Clark, is that the database really says more than it seems to at first sight. He adds an extra symbol, $=$, and extends the database to a *completed database* (CDB) by converting all clauses to "iff" assertions and adding some equality axioms; see [5], or [16, p. 70], or Section 5 below. Some problems involved with the CDB are discussed in [8].

Another idea is Reiter's *domain closure axiom* (DCA) [22], which says that the Herbrand universe (the set of all closed terms) is everything there is. However, if we take DCA literally, we are led to a semantics which is highly nonconstructive. One may call a semantics *constructive* iff the set of consequences of any finite database is a recursively enumerable set, so that in principle there is some complete deductive system for the semantics. It is a common phenomenon in logic that a semantics which is based on a fixed infinite domain of discourse will often be very far from constructive; in the context of logic programming, this phenomenon has been pointed out in [1], [3], and [7].

See [23] and [24] for a comparison of Clark's and Reiter's approaches. A common drawback of both of approaches is that they do not allow for the possibility that the truth value of a query might be undefined because the interpreter fails to halt. In 1938, Kleene (see [10] or [11]) suggested the use of three-valued logic to handle nontermination of computations, and this idea was applied in the context of logic programming by Lassez and Maher [14] and by Mycroft [19] to handle the problem of negative queries to pure Horn databases. Building on this, Fitting [7] gives a three-valued semantics for databases containing generalized program clauses. His approach may be viewed as adapting the DCA to a three-valued setting. He builds a three-valued least fixed-point semantics by transfinite recursion; as he points out, the recursion can, in the worst case, proceed through all the recursive ordinals and result in a semantics which is highly nonconstructive. As we point out in Section 8, this problem can occur even with pure Horn databases as soon as we allow negative queries (equivalently, positive queries to pure Horn databases could fail arbitrarily far out in the recursive ordinals).

Our semantics is constructive, and we present two equivalent definitions of it. The first one goes through a construction similar to that of [7], but cuts it off arbitrarily at stage ω. Here, although we follow DCA by working on only the Herbrand universe, constructiveness is achieved by the cutoff. This definition makes it easy to see what the semantics actually says about specific databases, but perhaps the cutoff may seem to be merely a "trick" to achieve constructiveness. The other definition uses Clark's CDB, but considers three-valued rather than two-valued models of it. Here, we achieve constructiveness by allowing the models to have domains of discourse extending the Herbrand universe.

5. THE COMPLETED DATABASE

We describe this in more detail, since it is basic for our semantics as well. The CDB may be thought of as a set of equality axioms, which do not depend on the database, plus a set of *completed definitions* derived from the database.

The equality axioms are expressed in the language of first-order logic with equality ($=$). The language has a countably infinite set of constant symbols and, for each n, countably infinitely many n-place function symbols. To simplify discussions

of syntax, we consider constants to be 0-place function symbols. The language has no predicate symbols other than $=$. In this language, define *Clark's equational theory* (CET) to be the usual axioms of predicate logic with equality (asserting that $=$ is an equivalence relation and that equals may be substituted for equals within any function), together with a set of axioms which we shall call the *freeness axioms*, because they assert that no objects are equal unless they are required to be equal by logic. More formally, the freeness axioms are the universal closures of the following three types of formulas. First:

$$f(X_1, \ldots, X_n) \neq g(Y_1, \ldots, Y_m)$$

whenever f and g are distinct function symbols of n and m places, respectively; n and/or m may be 0 (in which case we omit the parentheses following the function symbol). Second:

$$X \neq \tau$$

whenever τ is any term which contains the variable X. Third, we assert that each function is 1-1; that is, if f is n-place we have

$$f(X_1, \ldots, X_n) = f(Y_1, \ldots, Y_n) \rightarrow X_1 = Y_1 \wedge \cdots \wedge X_n = Y_n.$$

We now consider *models* for CET. Since CET contains the usual axioms for equality, we may consider only models in which $=$ is interpreted as true identity. By the *standard model*, we mean just the Herbrand universe. The domain of discourse of this model is the set of all ground terms, and function symbols have the obvious interpretation. In view of the distinctness axioms, *every* model for CET contains an isomorphic copy of the standard model (see [16, Exercise 10, p. 88]), so we shall consider only models which extend the standard model. It is easily verified that

Lemma 5.1. Let τ_1 and τ_2 be any terms. If τ_1 and τ_2 are unifiable, then

$$\text{CET} \vdash \exists(\tau_1 = \tau_2).$$

If they are not unifiable, then

$$\text{CET} \vdash \forall(\tau_1 \neq \tau_2).$$

PROOF. The first statement is obvious; the proof uses just the usual equality axioms, not the distinctness axioms. The second statement is easily proved by induction, using Robinson's inductive test (algorithm) for unifiability. □

This lemma is actually a special case of the deeper result (Theorem 5.3), that CET is complete and decidable. By completeness, if ϕ is any sentence in the language of CET, then it is provable from CET (i.e., true in all models of CET) iff it is true in the standard model. Decidable means that there is an algorithm for testing whether ϕ is provable from CET. Of course, this follows from completeness; simultaneously searching for a proof of ϕ and $\neg\phi$ provides an extremely inefficient algorithm; but in fact we shall outline a more reasonable decision procedure based on quantifier elimination.

We emphasize that $=$ is not a symbol used in the language of the database. If we wish to use an equality in a PROLOG database, we use a different symbol; e.g., "eq", with the database containing the axiom eq(X, X).

We now discuss *completed definitions* in the context of extended program clauses. Recall (Section 2) that we may consider all such clauses to be of the form

$$p(\tau_1,\ldots,\tau_n) :- \phi,$$

where ϕ is a formula of predicate logic. The *normalization* of this clause is defined to be

$$p(X_1,\ldots,X_n) :- \exists Y_1 \cdots \exists Y_j (X_1 = \tau_1 \wedge \cdots \wedge X_n = \tau_n \wedge \phi),$$

where $Y_1,\ldots,Y_j$ are the variables occurring in the clause and $X_1,\ldots,X_n$ are new variables.

If the n-place predicate p occurs m times in the head of a clause in DB, where $m > 0$, then the *completed definition* of p is

$$\forall X_1 \cdots \forall X_n (p(X_1,\ldots,X_n) \leftrightarrow \psi_1 \vee \cdots \vee \psi_m),$$

where each $p(X_1,\ldots,X_n) :- \psi_i$ is the normalization of a clause in DB. In the special case $m = 0$, the completed definition of p is

$$\forall X_1 \cdots \forall X_n (\neg p(X_1,\ldots,X_n)).$$

The *completed database*, CDB, is the set of axioms CET unioned with the completed definitions of all predicate symbols.

A very simple special case, which is sometimes sufficient to contrast our approach with others, occurs when all clauses in the database are propositional. If p is a 0-place predicate symbol (i.e., a proposition letter), and p occurs as the head of clauses $p :- \phi_i$ for $i = 1,\ldots,m$, and the ϕ_i are all ground, then the completed definition of p is just

$$p \leftrightarrow (\phi_1 \vee \cdots \vee \phi_m),$$

or just $\neg p$ if $m = 0$.

Observe that the above definitions of CDB assume that the database is finite, or at least that each predicate symbol occurs in the head of only finitely many clauses; otherwise, we would have to express the CDB using infinitary disjunctions. At first, this seems to be no serious problem, since real programs are finite anyway. However, a finite database using variables has infinitely many ground instances. Let grnd(DB) be the set of all ground instances of clauses in DB. In ordinary resolution logic (with the standard semantics), it is well known that DB has a model iff grnd(DB) has a model, so that DB and grnd(DB) may be treated equivalently; in fact completeness theorems about resolution are usually proved by lifting ground proofs. This works becauses all assertions in DB are purely universal. However, PROLOG-style semantics involves existential quantifiers as well, so that the equivalence of DB with grnd(DB) is problematical. For example, if DB contains just the clause "$p :- q(X)$", the completed definition of p is equivalent to

$$p \leftrightarrow \exists X q(X),$$

whereas in grnd(DB) the completed definition of p would have to be the infinitary statement

$$p \rightarrow \bigvee \{ q(\tau) : \tau \text{ is a ground term} \}.$$

Using such infinitary statements amounts to taking Reiter's domain closure axiom literally, and will lead to a semantics which is highly noneffective. This is one way to

view the construction in [7]; see also our Theorem 6.6 and Section 8. Our semantics is effective because we allow models in which these infinitary statements need not hold.

We now turn to the claimed completeness of CET. Perhaps the quickest proof would be to use the method of saturated models (see Section 5.4 of [4]). That is, it is quite easy to verify that CET has only infinite models, and that any two saturated models of the same cardinality are isomorphic. Decidability follows from completeness.

A more elementary (and somewhat more complicated) proof of completeness and decidability proceeds by quantifier elimination. CET itself does not admit quantifier elimination [for example, $\exists X(Y = f(X))$ is not equivalent to a quantifier-free (or even universal) property of Y], but quantifier elimination can be achieved if one adds symbols for the inverses of all the functions. More precisely, for each n-place function symbol f with $n > 0$, and each $i = 1, \ldots, n$, adjoin to the language a 1-place function inv_{fi} with the axioms

$$f(X_1, \ldots, X_n) = Y \rightarrow \text{inv}_{fi}(Y) = X_i$$

and

$$\left(\neg \exists X_1 \cdots X_n (f(X_1, \ldots, X_n) = Y)\right) \rightarrow \text{inv}_{fi}(Y) = Y.$$

The point of the second axiom is that if Y is not in the range of f, then we are coding this fact by setting $\text{inv}_{fi}(Y) = Y$. Note that if Y is in the range of f, then $\text{inv}_{fi}(Y) \neq Y$ by the freeness axioms in CET.

Call the resulting theory in the expanded language CET^+.

Lemma 5.2. CET^+ *is a conservative extension of* CET, *and every formula in the language of* CET^+ *is provably equivalent (from the axioms* CET^+*) to a quantifier-free formula.*

PROOF. By "conservative extension," we mean that every sentence in the language of CET which is provable from CET^+ is in fact provable from CET. This follows from the fact that CET^+ is an extension by definitions; see Section 4.6 of [25].

To prove quantifier elimination, note that any formula of the form $\exists X \phi(X)$, with ϕ quantifier-free in the language of CET^+, is equivalent to a finite disjunction of formulas of the form $\phi(\tau)$, where τ is a term constructed from the variables occurring in ϕ and the function symbols in the language of CET^+. See Section 5.5 of [25] for more on this sort of argument. □

Theorem 5.3. CET *is complete and decidable.*

PROOF. CET^+ decides the truth of all equations between ground terms and hence of all quantifier-free sentences, and using quantifier elimination, any sentence in the language of CET^+ can be reduced to a quantifier-free sentence. □

As an example of the quantifier elimination, $\exists Y(f(X, Y) = Z)$ is equivalent to $f(X, \text{inv}_{f2}(X)) = Z$. Also, if g is 2-place, then

$$\exists X\left(f\left(\text{inv}_{g1}(X), Y\right) = Z\right)$$

is equivalent to

$$f\left(\text{inv}_{g1}\left(g\left(\text{inv}_{f1}(Z), c\right)\right), Y\right) = Z;$$

the c here could be replaced by any other term; the point is that $f(U, Y) = Z$ forces $U = \text{inv}_{f1}(Z)$, and $U = \text{inv}_{g1}(X)$ is satisfied whenever $X = g(U, c)$.

A related analysis of terms is discussed by Lassez and Marriott [15], but in a different framework. Their Herbrand universe was built from a fixed finite list of function and constant symbols (in which case CET$^+$ would not be complete), whereas we are assuming infinitely many such symbols.

6. FORMAL SEMANTICS

We attempt to motivate our definition by some well-known examples. The following two simple propositional databases point out the problems with the CDB approach; similar problems arise with Reiter's approach (see [23] and [24]). Let DB1 be

$p :- q, \text{not}(q).$

$q :- q.$

Let DB2 be

$p :- q.$

$p :- \text{not}(q).$

$q :- q.$

In DB1 the completed definition of p is $p \leftrightarrow q \wedge \neg q$, while in DB2 the completed definition of p is $p \leftrightarrow q \vee \neg q$. In both, the completed definition of the proposition letter r is simply $\neg r$. Now, both CDBs predict an answer of "yes" to the query "?- not(r)", which is exactly what any PROLOG interpreter will give; but CDB1 predicts a "yes" to "?- not(p)", and CDB2 predicts a "yes" to "?- p," whereas any implementation which at all resembles PROLOG will fail to halt with queries involving p. There is an "out" here; we could simply say that this is just an example of incompleteness of the interpreter. However, although we must expect incompleteness eventually, it is strange that it must manifest itself with such trivial databases. Furthermore, this "out" is denied us by DB3:

$p :- \text{not}(p).$

Here, the CDB is $p \leftrightarrow \neg p$, which is inconsistent, and thus has all statements as logical consequences, so that any interpreter which gave any answer to "?- r" would be sound by this definition.

Instead, we take the approach of three-valued logic. We thus have three primitive truth values, **t**, **f**, and **u** (true, false, and undefined), with the truth tables those given by Kleene (see [10] or Section 64 of [11]); his motivation was also the description of computations which failed to return an answer. In all three examples, our semantics will imply that p has truth value **u**, which means that any sound interpreter must return "unclear" for queries "?- p" or "?- not(p)", whereas the letter r will have truth value **f** as expected.

Kleene's truth tables can be summarized as follows. If ϕ is a Boolean combination of the atoms, **t**, **f**, and **u**, its truth value is **t** iff *all* possible ways of putting in **t**

or **f** for the various occurrences of **u** lead to a value **t** being computed in ordinary 2-valued logic; ϕ gets value **f** iff $\neg\phi$ gets value **t**, and ϕ gets value **u** otherwise.

These truth values can be extended in the obvious way to predicate logic, thinking of the quantifiers as infinite conjunctions or disjunctions. Thus, $\exists X\phi(X)$ has truth value **t** iff there is at least one element in the domain of discourse at which ϕ is **t**, it has value **f** iff it is **f** at all elements, and it is **u** otherwise. Likewise for universal quantifiers.

We shall describe two equivalent definitions of our formal semantics. The first one uses a construction similar to that of the "least Herbrand model", as in [26], adapted for 3-valued logic in [7].

A *partial interpretation* is defined to be any (total) function I from the set of all ground atoms into $\{\mathbf{t},\mathbf{f},\mathbf{u}\}$. There is a natural notion of extension: $I \leq J$ (J extends I) iff for all ground atoms α, if $I(\alpha)$ is **t** or **f**, then $J(\alpha) = I(\alpha)$. We all I *total* iff $I(\alpha)$ is never **u**.

A partial interpretation I is extended in a natural way to a function $\hat{I}$ defined on all sentences (formulas with no free variables); so for any sentence ϕ of predicate logic, $\hat{I}(\phi)$ is either **t**, **f**, or **u**. This extension is obtained by following Kleene's truth tables for 3-valued logic; in line with the domain closure axiom (see Section 4), we are considering our domain of discourse to be just the Herbrand universe. Thus, for existential quantifiers, $\hat{I}(\exists X\phi(X)) = \mathbf{t}$ iff there is at least one closed term τ for which $\hat{I}(\phi(\tau)) = \mathbf{t}$; the value is **f** iff for all closed τ one has $\hat{I}(\phi(\tau)) = \mathbf{f}$; the value is **u** otherwise. For universal quantifiers, $\hat{I}(\forall X\phi(X)) = \mathbf{f}$ iff there is at least one closed term τ for which $\hat{I}(\phi(\tau)) = \mathbf{f}$; the value is **t** iff for all closed τ one has $\hat{I}(\phi(\tau)) = \mathbf{t}$; the value is **u** otherwise.

Now, we can view I as some partial information about the state of the world, and the database DB as a method of extending I to some further information ext(I), which is the partial interpretation J such that for each ground atom α:

(1) $J(\alpha) = \mathbf{t}$ iff for *some* clause

$$\beta :- \phi$$

in the database, $\alpha = \beta\sigma$ for some ground substitution σ such that

$$\hat{I}(\phi\sigma) = \mathbf{t}.$$

By σ being ground, we mean that it replaces all variables in β as well as all free variables in ϕ by ground terms [otherwise $\hat{I}(\phi)$ would not be defined].

(2) $J(\alpha) = \mathbf{f}$ iff for *all* clauses

$$\beta :- \phi$$

in the database, and all ground substitutions σ, if $\alpha = \beta\sigma$, then

$$\hat{I}(\phi\sigma) = \mathbf{f}.$$

(3) $J(\alpha) = \mathbf{u}$ otherwise.

The function ext is not increasing [i.e., $I \leq \text{ext}(I)$ need not hold], but it is monotone [i.e., $I \leq J$ implies $\text{ext}(I) \leq \text{ext}(J)$], so that it has a least fixed point, I_∞. This I_∞ may be constructed in the standard way by transfinite recursion: let I_0 be identically **u**, let $I_{\mu+1}$ be ext(I_μ), and take limits at limit ordinals; more precisely, if μ is a limit ordinal and α is a ground atom, there must be an ordinal $\xi < \mu$ such that

the values $I_\eta(\alpha)$ are constant for $\xi < \eta < \mu$, and we take this constant value for the value of $I_\mu(\alpha)$. If we work in pure Horn logic, the set of true ground atoms stabilizes at ω and corresponds to the usual minimal interpretation. However, even for pure Horn databases, the closure ordinal can be larger than all recursive ordinals, and the least fixed point can be a complete Π^1_1 set, since the truth values of atoms can switch from **u** to **f** past stage ω. For more on this, see Section 8, or see [7], which develops a semantics equivalent to our I_∞.

The suggestion is made in [7] that we restrict the class of PROLOG programs to avoid such a nonconstructive semantics. One possibility mentioned there is that we accept only programs for which the operator ext is continuous. The problem with this is that even some very simple pure Horn constructions [such as married(X) :– spouse(Y, X)] lead to discontinuous ext operators (even though the corresponding 2-valued Van Emden–Kowalski extension operator [26] is continuous for pure Horn databases). Of course, a discontinuous ext does not guarantee that the closure ordinal will be $> \omega$, and [7] also suggests that we look for conditions weaker than continuity which would guarantee closure by stage ω. However, any such restriction would rule out many useful programs. For example, consider an interpreter for a language such as PROLOG (or BASIC or LISP). If we write such an interpreter in PROLOG, there are natural queries whose truth is not decided until after stage ω. For example, say our database defines "term(P, X)" to mean that program P terminates with input X. By the unsolvability of the halting problem plus the effectiveness of our semantics (see Theorem 6.2 below), there must be programs, prog, which in fact fail halt with all input values, so that a "no" answer is supported by I_∞ for the query "?- term(prog, X)", but for which $\exists X$term(prog, X) does not get value **f** until stage $\omega + 2$.

Instead, we chop this recursion off at ω anyway in the definition of our semantics. If ϕ is any sentence, we say that ϕ is *supported* by our semantics iff $\hat{I}_n(\phi) = \mathbf{t}$ for some finite n. Thus, in the example just given, a "no" answer is *not* supported. We shall show (Theorem 6.2) that our semantics is constructive. Terminating the recursion in this way may seem artificial, but we shall show (Theorem 6.3) that the same semantics is obtained in a more natural way by considering three-valued models of the CDB.

Observe that for the three examples given at the beginning of this section, $I_\infty = I_1$, and the truth values of the letters are as advertised. More generally,

Lemma 6.1. If the database is finite and consists of n ground clauses, then $I_\infty = I_{n+1}$.

We now show that in our semantics, the set of consequences of the database is recursively enumerable; by standard undecidability results, we cannot expect it to be recursive.

Theorem 6.2. If the database if finite, then for each finite n, $\hat{I}_n$ *is decidable, and the set of all supported queries is recursively enumerable.*

PROOF. Our inductive definition of the I_n yields a computable function which assigns, to each sentence ϕ, natural number n, and truth value **v**, a sentence value(ϕ, n, **v**), in the language of CET, which is true in the standard model of CET

(see Section 5) iff $\hat{I}_n(\phi) = \mathbf{v}$. The theorem is now immediate from the decidability of CET (Theorem 5.3). □

In applications, we should think of this theorem being used with ϕ of the form $\forall(\psi\sigma)$, where ψ is the user's query to the database and σ is PROLOG's answer [which should signify that $\forall(\psi\sigma)$ follows from the database]. So, the set of logically correct answers to a query is recursively enumerable.

One must be careful to distinguish our semantics from $\hat{I}_\omega$. If α is an atomic *sentence*, then α is supported by our semantics iff $I_\omega(\alpha) = \mathbf{t}$. However, there are examples of atomic *formulas* ψ such that $\forall\psi$ is true at stage ω but not at any finite stage (see Section 8); that is, all ground instances of ψ are supported by our semantics but $\forall\psi$ is not. The set of atomic ψ such that $\hat{I}_\omega(\forall\psi) = \mathbf{t}$ is not in general r.e., and in the worst case can be a complete Π^0_2 set.

We now turn to the second approach to obtaining our semantics—namely, via 3-valued models for CDB. Roughly, a sentence is supported by our semantics iff it is a semantic consequence of CDB in 3-valued logic, but we must clarify two points about 3-valued models. First, we intend that in such models the equality is 2-valued. Thus, our models all consist of a base set, plus interpretations of the various functions in the usual 2-valued manner; $=$ is interpreted as identity as usual, and we demand that the equality axioms CET be valid. Second, we must be careful about the propositional connective $\leftrightarrow$. Kleene's 3-valued logic has the value of $\mathbf{u} \leftrightarrow \mathbf{u}$ computed as $\mathbf{u}$, and this is correct for the symbol $\leftrightarrow$ occurring within an extended program or query clause, but it is wrong for the $\leftrightarrow$ added in the CDB. To avoid confusion, we demand that if extended program or query clauses use an "iff", they write it as $\equiv$, and we reserve $\leftrightarrow$ for use in the CDB. Then $\equiv$ has Kleene's 3-valued truth table, whereas $\phi \leftrightarrow \psi$ will have value $\mathbf{t}$ iff ϕ and ψ have the same value, and $\mathbf{f}$ otherwise (this truth table was suggested by Łukasiewicz in 1920; see [11]). So, for example, if the database has proposition letter p in the head of the one clause,

$$p :- (q \equiv q),$$

then the completed definition of p is $p \leftrightarrow (q \equiv q)$. For this to be true in a 3-valued model for CDB, it is necessary that q is $\mathbf{u}$ and p is $\mathbf{u}$ or that q is not $\mathbf{u}$ and p is $\mathbf{t}$. That is, if q gets a definitive truth value than p must be true; otherwise, the computation of p cannot produce an answer.

Theorem 6.3. Assume that the database is finite, and let ϕ be any sentence. Then the following are equivalent:

(1) *ϕ has value $\mathbf{t}$ is every 3-valued model for* CDB.

(2) *ϕ is supported by our semantics.*

PROOF. That (2) implies (1) is easily proved by induction on the stage at which ϕ becomes true. In proving (1) implies (2), we shall actually produce *one* 3-valued model, N, for CDB, such that for every sentence ϕ, if ϕ is $\mathbf{t}$ in N, then $\hat{I}_n(\phi) = \mathbf{t}$ for some finite n. N is obtained by an ultrapower construction.

Let U be a countably incomplete ultrafilter on some index set S. Let M_n be the (3-valued) model whose base set is the set of ground terms, and whose truth values

are determined by I_n, and let N_n be the ultrapower $(M_n)^S/U$. All the N_n have the same base set, and they all interpret function symbols the same way. For each formula, ψ, with free variables $X_1, \ldots, X_k$, let us use $\text{val}(\psi, a_1, \ldots, a_k, N_n)$ to denote the truth value of ψ in N_n at the k-tuple of elements $a_1, \ldots, a_k$. This cumbersome notation is necessitated by the fact that, unlike in the Herbrand model case, not all elements of N_n are the denotations of terms. Observe that the N_n *cohere* in the sense that if **v** is either **t** or **f** and $\text{val}(\psi, a_1, \ldots, a_k, N_n) = \mathbf{v}$, then for all $m > n$, $\text{val}(\psi, a_1, \ldots, a_k, N_m) = \mathbf{v}$ also; this holds because the M_n cohere and the ultrapower produces elementary equivalent structures. We may thus define the limit, N, of the N_n. This model has the same base set as the N_n, with truth values computed as follows. For each *atomic* formula ψ with variables $X_1, \ldots, X_n$, for each $a_1, \ldots, a_k$ in the model, and for each truth value **v**, we set $\text{val}(\psi, a_1, \ldots, a_k, N) = \mathbf{v}$ if the $\text{val}(\psi, a_1, \ldots, a_k, N_n)$ are **v** for all n larger than some n_0.

Observe that N is similar to our I_ω. However, unlike I_ω, we have the following *continuity property*: If ψ is any formula, **v** a truth value, and $\text{val}(\psi, a_1, \ldots, a_k, N) = \mathbf{v}$, then $\text{val}(\psi, a_1, \ldots, a_k, N_n) = \mathbf{v}$ for all n larger than some n_0. For ψ atomic, continuity holds by definition, and we may prove continuity in general by induction on complexity. The induction passes easily over propositional connectives. To illustrate the step for quantifiers in the place where it is nontrivial, assume that ψ has free variable X, that ψ is $\exists Y \chi$, and that continuity has been proved for χ. Now suppose that **u** is the eventual value of the $\text{val}(\psi, a, N_n)$, so that for each n, we have $\text{val}(\phi, a, N_n) = \mathbf{u}$. Then for each b in the base set and each n, $\text{val}(\chi, a, b, N_n)$ must be **f** or **u**, with at least one b_n having $\text{val}(\chi, a, b_n, N_n) = \mathbf{u}$. Since the N_n cohere, we have also $\text{val}(\chi, a, b_n, N_m) = \mathbf{u}$ for all $m < n$. But now, since the ultrapower yields a saturated model, we may choose a b such that for all m, $\text{val}(\chi, a, b, N_m) = \mathbf{u}$. Then, by continuity for χ, $\text{val}(\chi, a, b, N) = \mathbf{u}$. Again by continuity for χ, there is no c for which $\text{val}(\chi, a, c, N) = \mathbf{t}$. Thus, $\text{val}(\psi, a, N) = \mathbf{u}$.

By continuity, if ϕ is any sentence with value **t** in N, then ϕ must have value **t** in some N_n, which implies, since the ultrapower defines an elementary embedding, that $I_n(\phi)$ is **t**. Also, using continuity, it is easy to show that N is indeed a 3-valued model for CDB. □

2-valued models are a special case of 3-valued models, so we now have proved that our semantics is more restrictive than Clark's CDB semantics.

Corollary 6.4. Assume that the database is finite and ϕ is any sentence. If ϕ is supported by our semantics, then ϕ is a logical consequence in 2-valued logic of CDB.

The converse of this corollary is false, as the examples in the beginning of this section show. One can verify, however, that in the case of pure Horn logic, where the semantics was never problematical, the various approaches to semantics all agree.

Theorem 6.5. Assume the database is finite and pure Horn, and assume ϕ is a pure Horn query clause (that is, a conjunction of atoms). Then the following are

equivalent:

(1) $\forall\phi$ *is a (2-valued) semantic consequence of the database.*

(2) $\forall\phi$ *is a (2-valued) semantic consequence of* CDB.

(3) $I_\infty(\forall\phi) = \mathbf{t}$.

(4) $\forall\phi$ *is supported by our semantics.*

I_∞ corresponds to models which allow only the Herbrand universe as elements. The following is the appropriate analogue of Theorem 6.3, and is much easier to verify.

Theorem 6.6. Assume the database if finite and ϕ *is any sentence. Then the following are equivalent*:

(1) ϕ *has value* **t** *in any 3-valued model for* CDB *whose domain of discourse is just the set of closed terms.*

(2) $I_\infty(\phi) = \mathbf{t}$.

As in other approaches to PROLOG semantics, we may easily verify that the translation in [17] from extended program clauses to generalized program clauses (called their *general form*; see p. 229) is correct. Essentially the same proof as in [17] works; namely show by induction that:

Theorem 6.7. Let DB *be a finite database consisting of extended program clauses, and form the database* DB′ *by replacing each clause by its general form. Use* DB *to form the* I_μ *and* DB′ *to form the* I'_μ. *Then for each formula* ϕ, *there is a finite n such that whenever* $\phi\sigma$ *is any ground instance of* ϕ, μ *is any ordinal, and* **v** *is* **t** *or* **f**, $\hat{I}_\mu(\phi\sigma) = \mathbf{v}$ *iff* $\hat{I}'_{\mu+n}(\phi\sigma) = \mathbf{v}$.

[17] does not actually use the propositional connective $\equiv$ in extended program clause, but it can easily be included (there and here), and then replaced in the standard way by $\rightarrow$ and $\wedge$ in the passage to generalized program clauses.

Observe that in our three-valued semantics, an extended query clause which is a classical (two-valued) tautology may fail to be supported; this can happen even when the query is itself in the database or the CDB. For example, suppose the database consists of the single clause "r :– r"; then the CDB contains $r \leftrightarrow r$ plus the negations of all other proposition letters. Then $I_\omega(r) = \mathbf{u}$, which gives $r \leftrightarrow r$ value **t**, but the queries "?- r :– r" and "?- $r \equiv r$" (note the different *iff* in the query clause) have value **u**, not **t**. At first sight this seems pathological, but it corresponds with the fact that the translations of these queries into standard PROLOG (e.g., "?- (not(r); r)") will cause an infinite loop with this database.

7. ANSWER SUBSTITUTIONS

Elsewhere in this paper we have attempted to contrast our approach with other approaches. For example, in Section 6 we presented some simple ground databases

in which a query is supported by the CDB, but not by our semantics. In Section 8, we will point out the noneffectiveness that can occur if we consider I_μ for infinite μ.

In this section, we consider a class of examples where all the standard approaches to semantics agree. Here, effectiveness is *not* a problem, as $I_\infty = I_n$ for some small n. In these examples, also, our approach and the CDB approach agree. Our intent is to illustrate the complexity of the computed answer substitutions, and therefore the difficulty of constructing a practical PROLOG interpreter which is complete for our semantics, or, for that matter, for any reasonable semantics. As a simple example, consider

q :– not $p(X)$.
$p(X)$:– isc(X).
$p(X)$:– nonc(X).
isc(c).
nonc(X) :– not isc(X).

This database is *finitely determinate*; that is, for some finite n ($= 4$ here), $I_\infty = I_n$, and every ground atom receives value **t** or **f**. In our (and the CDB) semantics, $p(X)$ is true of everything, since "isc" is true of "c" and "nonc" is true of everything but "c". Hence, an answer of "no" is supported to "?- q". Any term other than "c" could be correctly returned as an answer to the query "?- nonc(X)"; a *complete* interpreter presumably should return the answer "$X \neq c$".

This example illustrates the point is that even if we only care about our interpreter being complete for ground queries such as "?- q", such an interpreter would have to produce complete answers to nonground queries as intermediate steps. Thus, in any case, we are forced to design an interpreter which returns complete answers to nonground queries as well. These complete answers need not be just equations of the form "$X = \sigma$", as in ordinary PROLOG, but could be negations of these equations, or arbitrary Boolean combinations of these equations. In fact, given any set of terms which can be described within the theory CET (see Section 5), one can find a finitely determinate database for which this set is the complete answer.

An interpreter which would find answers in situations like this could operate like the following version of parallel PROLOG. Each intermediate goal is not terminated, but receives answer substitutions from its children. As these substitutions accumulate, they are not merely communicated to the parent, but they are amalgamated into a more general form if possible. Thus, the process working on "-? q" invokes a child working on "?- $p(X)$" and grandchildren working on "?- isc(X)" and "?- nonc(X)". When the "?- $p(X)$" process eventually receives an "$X = c$" and an "$X \neq c$" from its children, it amalgamates these and sends an $X = X$ (identity substitution) to its parent, which now can fail q. This amalgamation procedure is the general case is related to the quantifier elimination procedure for CET (see Lemma 5.2).

8. NONEFFECTIVENESS

Although our semantics only uses I_n for finite n, we remark here on the complexity of I_∞. We shall be fairly brief, since similar results are known in the literature, most

recently in [7]. The main point is that any semantics based on a fixed infinite domain of discourse (such as the Herbrand universe) is bound to have high logical complexity. Blair [3] was the first to point this out in the context of logic programming. In our semantics, the noneffectiveness can occur even with pure Horn databases, since atoms can switch in value from **u** to **f** arbitrarily far out in the recursive ordinals.

Recall that ω_1^{CK} denotes Church-Kleene omega one—ie., the first infinite ordinal not isomorphic to a recursive well-ordering of ω. The following fact is a special case of a much more general results on inductive closures, due to Barwise, Gandy, and Moschovakis; see [9], Chapter 6 of [2], or chapter 9 of [18].

Theorem 8.1. If DB *is finite (or infinite and recursive), then* $I_{\omega_1^{CK}} = I_\infty$, *and*

$$\{\phi : \phi \text{ is ground and } \hat{I}_\infty(\phi) = \mathbf{t}\}$$

is a Π_1^1 *set.*

Also, in view of the fact that our semantics is r.e. (Theorem 6.2),

Theorem 8.2. If DB *is finite, then*

$$\{\psi : \psi \text{ is atomic and } \hat{I}_\omega(\forall\psi) = \mathbf{t}\}$$

is a Π_2^0 *set.*

These two theorems are best possible; this can be seen by exploring the fact that as we continue the iteration into infinite ordinals, it becomes possible to express arbitrary quantification over the natural numbers. More precisely, let us use s as a 1-place function and let us use 0 as a constant. For each natural number n, let $\underline{n}$ represent the closed term formed by applying s n times to 0. In this way, our PROLOG databases can talk about natural numbers without departing from pure logic (such a treatment of natural numbers is the official one in IC-PROLOG [6]). Before we state any theorems, we consider the following very simple example:

number(0).

number($s(X)$) :– number(X).

void($s(X)$) :– void(X).

The predicate number(X) defines the set of terms representing natural numbers. The predicate void(X) becomes **f** for all terms by stage ω, so $\hat{I}_\omega(\forall X \neg \text{void}(X)) = \mathbf{t}$, so a semantics based on I_∞ would support a "no" answer to "?- void(X)". Real PROLOG will return an infinite loop with this query, corresponding to the fact that $\hat{I}_k(\forall X \neg \text{void}(X)) = \mathbf{u}$ for all finite k.

Following Blair [3], we can represent all recursive and r.e. sets of natural numbers by pure Horn databases. He shows that if P is any recursive set, there is a finite pure Horn database involving the 1-place predicate p (plus other predicates and functions) such that for each natural number n, $I_\omega(p(\tau))$ is **t** when τ is of the form $\underline{n}$ for $n \in P$, and **f** for τ any other closed term. Analogous results hold if P is any recursive predicate in several variables. The proof in [3] uses the fact that one can simulate a Turing machine in pure Horn logic; one could also use induction on the

construction of the recursive functions by primitive recursion, composition, and the μ operator. We remark that P cannot be an arbitrary recursive set of closed terms, since for pure Horn databases, if $p(c)$ is **t** for a constant c not mentioned in the database, then $p(\tau)$ must be **t** for all closed terms.

Recursively enumerable sets of natural numbers can be represented in the following weaker sense:

Lemma 8.3. If Q is any r.e. set of natural numbers, then there is a finite pure Horn database involving the 1-place predicate q such that for each natural number n, $I_\omega(q(\underline{n}))$ is **t** *for $n \in Q$ and* **u** *otherwise; and $I_\omega(q(\tau))$ is* **f** *for closed terms τ which do not represent natural numbers.*

PROOF. Suppose that Q is the projection of the primitive recursive 2-place predicate P. Add the clause "$q(X)$:– number(X), $p(X, Y)$." to a database which represents P by p and contains the above definition of "number". □

Of course, similar results hold when Q is an r.e. predicate in several variables.

Theorem 8.4. Let R be any Π_2^0 set of natural numbers. Then there is a finite database using generalized program clauses such that for any natural number n, $I_\omega(\forall Y q(\underline{n}, Y)) = \mathbf{t}$ iff $n \in R$.

PROOF. Suppose $n \in R$ iff $(n, m) \in Q$ for all m, where Q is r.e. A (pure Horn) database representing Q almost works, but $q(\underline{n}, Y)$ will be false when Y is instantiated to a term which does not represent a number, so add to such a database the definition of "number" above plus:

$q(X, Y)$:– notanumber(Y).

notanumber(Y):– not(number(Y)). □

The use of "not" in this database seems trivial, but cannot be avoided in view of Theorem 6.5. However, even for pure Horn databases, the complexity of I_∞ can be large, since any Π_1^1 set can be represented as the set of n for which I_∞ supports a "yes" answer to "?- not($r(\underline{n})$)".

Theorem 8.5. Let R be any Π_1^1 set of natural numbers. Then there is a finite pure Horn database such that for any natural number n, $I_\infty(r(\underline{n})) = \mathbf{f}$ iff $n \in R$.

PROOF. By the Suslin-Kleene analysis, there is a primitive recursive total ordering T, of the natural numbers and a primitive recursive set Q of pairs, such that for each n, $n \in R$ iff $Q_n = \{m : (n, m) \in Q\}$ is well ordered by T. We may choose T and Q so that 0 is the top element in the order T, and so that for all n, $(n, 0) \in Q$. Add the following to a database that represents T and Q by t and q:

$r(X)$:– $p(X, 0)$.

$p(X, Y)$:– $t(V, Y), q(X, V), p(X, V)$.

By stage ω, truth values for t and q will be completely determined. Note that $r(\underline{n})$

and $p(\underline{n}, \underline{m})$ can never get value **t**. $p(\underline{n}, \underline{m})$ will get value **f** at a stage $\mu > \omega$ iff by this stage, $p(\underline{n}, j)$ already has value **f** for all j such that jTm and $j \in Q_n$. Thus, $p(\underline{n}, 0)$ and $r(\underline{n})$ will eventually become **f** iff Q_n is well ordered. If Q_n is not well ordered, then $p(\underline{n}, \underline{m})$ will eventually become **f** only for m in the well-ordered initial part of Q_n; for other m, including $m = 0$, the truth value will remain **u** forever. □

The closure ordinal order such a database must be exactly ω_1^{CK} unless R is hyperarithmetical. By modifying this example, one may, as in [3], construct finite databases whose closure ordinal is any recursive ordinal, or whose complexity is any desired level in the Kleene hierarchy.

9. PROLOG INTERPRETERS

Since our semantics is based upon what we expect PROLOG to do, we should investigate whether it in fact corresponds to the answers returned by a PROLOG-style interpreter. In the best case, we would like a completeness theorem, but at least we should verify that the standard approaches to PROLOG are sound.

If we restrict ourselves to finite propositional databases, everything works perfectly. We can describe a query evaluation procedure that returns "yes" if and only if the query is supported. As in the case even with pure Horn logic, we must allow this procedure to be nondeterministic. There are two possible places for nondeterminism: the first is in the choice of the ordering of the clauses, and the second is in the choice of the ordering of the literals in each clause; these places correspond to OR parallelism and AND parallelism in a parallel implementation. In pure Horn logic, only the first nondeterminism is needed (see [5] or Section 9 of [16]); however, once we have added a "not", we need the second as well for completeness. For example, if the database is

$p :- q, r.$

$q :- q.,$

the query "?- not(p)" should return "yes", which requires the interpreter to evaluate the r before the q in the first clause.

Now, assume a finite database DB is given. We present our evaluation procedure in the form of pseudocode for a nondeterministic automaton; the points of nondeterminism are labeled by "CHOOSE". We call our main procedure eval; it is fed a list ϕ (possibly empty) of literals, and, if it halts, returns either "yes" or "no". Semantically, the list denotes its conjunction; in line with this, we identify the empty list with *true*.

For such an evaluation procedure, soundness means that under *any* choices in the nondeterminism, if "yes" is returned then ϕ (i.e., the conjunction of the list) gets value **t** in our semantics, and if "no" is returned, then ϕ gets value **f**. Since "yes" and "no" are the only possible outputs here, soundness requires that if ϕ has value **u**, then all choices must fail to halt. Completeness means that if ϕ gets value **t** in our semantics, then *some* choices lead to "yes" being returned, and if ϕ gets value **f**, then some choices lead to a "no".

If ϕ is such a list and λ is a literal on that list, then $\phi - \{\lambda\}$ denotes the list with the first occurrence of λ deleted. Our procedure has a subroutine, evallit, which computes the truth value of a literal.

```
function eval(φ).
  if φ is empty return ("yes").
  CHOOSE a literal, λ in φ
  if evallit(λ) = "no" return "no"
  if evallit(λ) = "yes" return eval(φ − {λ})

function evallit(λ)
  if λ is negative, ¬α then
    if evallit(α) = "yes" return "no"
    if evallit(α) = "no" return "yes"
  if λ is positive then
    CHOOSE an ordering, {ψ1,..., ψk} of all
    bodies of clauses in DB with head λ
  for i := 1 to k do
    if eval(ψi) = "yes" return "yes"
  return ("no")
    /* so, if k = 0, we return "no" */
```

Theorem 9.1. The interpreter eval *is sound and complete*

PROOF. Soundness is proved by induction on the number of recursive calls to eval. Completeness is proved by induction on the (finite by Lemma 6.1) stage at which ϕ receives truth value **t** or **f**. □

One might replace the nondeterminism by some sort of *fairness* assumption, as in Lassez and Maher [13], which guarantees that all choices eventually get searched. Both nondeterminism and fairness would, if actually implemented, have the effect of replacing the depth-first search of standard PROLOG by a breadth-first search.

It is easy to see how to modify this (in *theory*) to work in predicate logic; see the remarks at the end of Section 7. This would require the interpreter to keep track of arbitrarily complex descriptions of answers. One may view IC-PROLOG [6] and MU-PROLOG [20,21] as subsets of this ideal interpreter which return an error message when forced to deal with an answer more complicated than a simple substitution for variables, so these systems are sound but not complete in our semantics.

REFERENCES

1. Apt, K. R. and Van Emden, M. H., Contributions to the Theory of Logic Programming, *J. Assoc. Comput. Mach.* 29:841–863 (1982).
2. Barwise, J., *Admissible Sets and Structures*, Springer, Berlin, 1975.
3. Blair, H. A., The Recursion-Theoretic Complexity of the Semantics of Predicate Logic as a Programming Language, *Inform. and Control* 54:25–47 (1982).
4. Chang, C. C. and Keisler, H. J., *Model Theory*, North-Holland, Amsterdam, 1973.
5. Clark, K. L., Negation as Failure, in: H. Gallaire and J. Minker (eds.), *Logic and Databases*, Plenum, New York, 1978.

6. Clark, K. L., McCabe, F. G., and Gregory, S., IC-PROLOG Language Features, in: K. L. Clark and S.-A. Tärnlund (eds.), *Logic Programming*, Academic, New York, 1982.
7. Fitting, M. R., A Kripke-Kleene Semantics for Logic Programs, *J. Logic Programming* 2:295–312 (1985).
8. Flannagan, T., The Consistency of Negation as Failure, *J. Logic Programming* 3:93–114 (1986).
9. Gandy, R. O., Inductive Definitions, in: J. E. Fenstad and P. G. Hinman (eds.), *Generalized Recursion Theory*, North-Holland, Amsterdam, 1974, pp. 265–300.
10. Kleene, S. C., On Notation for Ordinal Numbers, *J. Symbolic Logic* 3:150–155 (1938).
11. Kleene, S. C., *Introduction to Metamathematics*, Van Nostrand, Princeton, 1952.
12. Kowalski, R. A., Algorithm = Logic + Control, *Comm. ACM* 2:424–436 (1979).
13. Lassez, J-L. and Maher, M. J., Closures and Fairness in Programming Logic, *Theoretical Computer Science* 29:167–184 (1984).
14. Lassez, J-L,. and Maher, M. J., Optimal Fixedpoints of Logic Programs, *Theore. Comput. Sci.* 39:15–25 (1985) (reprinted from FST-TCS Conference, Bangalore, 1983).
15. Lassez, J-L. and Marriott, K. G., Explicit Representation of Terms Defined by Counterexamples, IBM Research Report, T. J. Watson Research Center, 1986 (presented at FST-TCS Conference, New Delhi, 1986).
16. Lloyd, J. W., *Foundations of Logic Programming*, Springer, Berlin, 1984.
17. Lloyd, J. W. and Topor, R. W., Making Prolog More Expressive, *J. Logic Programming* 1:225–240 (1984).
18. Moschovakis, Y. N., *Elementary Induction on Abstract Structures*, North-Holland, Amsterdam, 1974.
19. Mycroft, A., Logic Programs and Many-Valued Logic, in Proceedings of Symposium on Theoretical Aspects of Computer Science, *Lecture Notes in Comput. Sci.* 166:274–286 (1984).
20. Naish, L., Automating Control for Logic Programs, *J. Logic Programming* 2:167–183 (1985).
21. Naish, L., The MU-Prolog 3.2 Reference Manual, TR85/11, Dept. of Computer Science, Univ. of Melbourne, 1985.
22. Reiter, R., On Closed World Data Bases, in: H. Gallaire and J. Minker (eds.), *Logic and Databases*, Plenum, New York, 1978, pp. 293–322.
23. Shepherdson, J. C., Negation as Failure, *J. Logic Programming* 1:51–79 (1984).
24. Shepherdson, J. C., Negation as Failure II, *J. Logic Programming* 2:185–202 (1985).
25. Shoenfield, J. R., *Mathematical Logic*, Addison-Wesley, Reading, Mass., 1967.
26. Van Emden, M. and Kowalski, R., The Semantics of Predicate Logic as a Programming Language, *J. Assoc. Comput. Mach.* 23:733–742 (1976).