SIGNED DATA DEPENDENCIES IN LOGIC PROGRAMS

by

Kenneth Kunen

Computer Sciences Technical Report #719

October 1987

SIGNED DATA DEPENDENCIES IN LOGIC PROGRAMS

Kenneth Kunen*

Computer Sciences Department
University of Wisconsin
Madison, WI 53706, U.S.A.
kunen@cs.wisc.edu

September 28, 1987

ABSTRACT

Logic programming with negation has been given a declarative semantics by Clark's Completed Database, CDB, and one can consider the consequences of the CDB in either 2-valued or 3-valued logic. Logic programming also has a proof theory given by SLDNF derivations. Assuming the data dependency condition of *strictness*, we prove that the 2-valued and 3-valued semantics are equivalent. Assuming *allowedness* (a condition on occurrences of variables), we prove that SLDNF is complete for the 3-valued semantics. Putting these two results together, we have completeness of SLDNF deductions for strict and allowed databases and queries under the standard 2-valued semantics. This improves a theorem of Cavedon and Lloyd, who obtained the same result under the additional assumption of *stratifiability*.

§**1. INTRODUCTION.** There have been many recent advances in elucidating the semantics of negation in logic programming, but some serious problems still remain.

To a first approximation, we may think of the semantics as given by Clark's Completed Database [3]. Given a database, DB, we form CDB, which consists of some equality axioms plus a completed definition of each predicate symbol; roughly, this completed definition is obtained by replacing the Prolog "if" by an "iff".

However, the CDB does not always capture the intended Prolog semantics. For example, if DB consists of the single clause, $p :- \neg p$, then the CDB is $p \leftrightarrow \neg p$, which is inconsistent, so all statements follow logically from it, whereas we would expect anything resembling Prolog to return "no" to $?- \ q$, "yes" to $?- \ \neg q$, and not to return at all from $?- \ p$. This sort of problem is solved by using 3-valued logic as described in [8] (which is based on earlier such approaches in [6, 11, 13]). This logic has three truth values, **t** (true), **f** (false), and **u** (undefined). We use the truth value **u**, as did Kleene, to capture the notion of a computation which fails to halt. Our CDB is verbatim the same as Clark's original definition, but we use its consequences in 3-valued logic (with Kleene's truth tables), rather than in 2-valued logic.

The 3-valued semantics is weaker than the 2-valued, in the sense that every query supported in the 3-valued semantics is also supported in the 2-valued semantics, but not

* This research was supported by NSF Grant DMS-8501521.

conversely. In 3-valued logic, the CDB is always consistent. In the example just considered, the 3-valued semantics forces q to be $\mathbf{f}$, $\neg q$ to be $\mathbf{t}$, and p to be $\mathbf{u}$, corresponding exactly to what Prolog does. In fact, in the *propositional* case, it is easy to prove a completeness theorem (see [8]), showing that any query clause is supported by the 3-valued semantics iff there is a Prolog-style derivation of it. The abstract notion of a Prolog-style derivation is essentially given by Clark's Query Evaluation Procedure [3], or, more formally, by the notion of an SLDNF derivation [12].

Although this 3-valued CDB approach seems like a good approximation to the intended semantics of negation in Prolog, it still leaves us with two major problems.

Problem 1: Incompleteness. In the non-propositional case, SLDNF is incomplete. Furthermore, as pointed out in [9], it is unlikely that any modification of SLDNF will lead to completeness unless we depart completely from the spirit of Prolog.

Problem 2: Many people will find 3-valued logic not as natural or easy to understand as 2-valued logic. This is not a mathematical problem, but it does indicate a failure to give programmers a clear and understandable explanation of the declarative meaning of their Prolog programs.

The purpose of this paper is to discuss sufficient conditions under which these problems go away. Regarding Problem 1, we shall show (Theorem 4.3) that the condition of *allowedness* (see [15, 1] or §2 – roughly, this says that every variable occurring in a clause must occur at least once within a positive literal in the body of that clause) is sufficient to guarantee completeness of SLDNF for the 3-valued semantics. Regarding Problem 2, we shall show (Theorerm 3.6) that the condition of *strictness* (see [1] or §2 – roughly, this says that no predicate depends on another predicate both positively and negatively) is sufficient to guarantee that the 3-valued and 2-valued semantics coincide.

If we put these results together, we have that, under the two conditions, allowedness and strictness, SLDNF is complete for the 2-valued semantics. This is an improvement of the earlier result of Cavedon and Lloyd [2], who proved completeness if one had these two conditions plus a third, *stratifiability*. [2] dealt directly with 2-valued models, and used the methods of Apt, Blair, and Walker [1] to construct 2-valued models of stratified databases. Stratifiability (defined in [1] or §2) says roughly that there are no cycles of data dependency through a negation. This notion seems to be great interest in its own right (see [10]), but turns out to be irrelevant here, although we shall see that the methods of [1] are useful even when dealing with non-stratified databases.

§2 will give precise definitions of the various syntactical notions used. §3 describes the basic methods in constructing models and converting 3-valued models to 2-valued models. §4 presents the completeness result mentioned above. §5 makes some remarks on possibilities for extending our results.

§2. SYNTAX.

We assume that our language for predicate logic is fixed in advanced, and contains, for each $n \geq 0$, a countably infinite set of n-place function symbols and a countably infinite set of n-place predicate symbols. 0-place function symbols are called constant symbols, and 0-place predicate symbols are called proposition letters. In addition, our language has a symbol, '=', for equality; this symbol never occurs in a database, but is used in forming the CDB.

A *literal* is an atomic formula or a negated atomic formula. A *program clause* is of the form

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

where α is atomic, $\lambda_1, \ldots, \lambda_n$ are literals, and $n \geq 0$; if $n = 0$, we write $\alpha :- \ true$, or just α. The database, DB, will always be a *finite* set of program clauses.

A *query clause* is of the form

$$\lambda_1, \ldots, \lambda_n \quad ,$$

where $n \geq 0$; if $n = 0$, we just write *true*. Of courses, actual queries will have $n \geq 1$, but it is useful to consider the case $n = 0$ when we consider intermediate goals in SLDNF derivations (see §4).

We now consider the data dependency relation. Let $PRED$ be the set of all predicate symbols. We use $\sqsupseteq$ to denote immediate dependency; thus, if $p, q \in PRED$, then $p \sqsupseteq q$ iff DB contains a clause in which p occurs in the head and q occurs in the body. Let $\geq$ denote the least transitive reflexive relation on PRED extending $\sqsupseteq$; so, $p \geq q$ means that p hereditarily depends on q. Of course, $\sqsupseteq$ and $\geq$ depend on DB, but there will be no danger of confusion here, since we shall only discuss one database at a time.

As does every transitive reflexive relation, $\geq$ has two associated relations on $PRED$. Let $\approx$ denote the equivalence relation defined by: $p \approx q$ iff $p \geq q$ and $q \geq p$. Let $>$ denote the transitive irreflexive relation defined by: $p > q$ iff $p \geq q$ and $q \not\geq p$.

If $\mathcal{P} \subseteq PRED$, we call $\mathcal{P}$ *downward closed* iff for all $p \in \mathcal{P}$ and $q \in PRED$, $q \leq p$ implies $q \in \mathcal{P}$. Note that this implies that $\mathcal{P}$ is a union of equivalence classes. Observe that $<$ is acyclic. When we get to semantics (§3), we shall build models for the CDB by induction on $<$, treating each equivalence class as a unit. In one step in the induction, we have interpretations defined for some downward closed set, $\mathcal{P}$, of predicate letters, and we extend this interpretation to an equivalence class minimal above $\mathcal{P}$.

When we discuss criteria which allow 3-valued models to be converted to 2-valued models, it will be important to consider whether predicates depend on other predicates positively or negatively. This signed dependency is defined as follows. We say $p \sqsupseteq_{+1} q$ iff there is a clause in DB with p occurring in the head, and q occurring in a *positive* literal in the body. We say $p \sqsupseteq_{-1} q$ iff there is a clause in DB with p occurring in the head, and q occurring in a *negative* literal in the body. Let $\geq_{+1}$ and $\geq_{-1}$ be the least pair of relations on $PRED$ satisfying:

$$p \geq_{+1} p$$

and

$$p \sqsupseteq_i q \ \& \ q \geq_j r \quad \Rightarrow \quad p \geq_{i \cdot j} r \ .$$

Note that by making $\geq$ and $\geq_{+1}$ reflexive, we always have $p \geq p$ and $p \geq_{+1} p$, so that, in the case of a singleton equivalence class, $\{p\}$, these partial orders do not distinguish whether or not p is defined recursively from itself. This distinction is often important, but turns out to be irrelevant for this paper. The important distinction for us will be whether $p \geq_{-1} p$ – i.e., whether p is defined negatively from itself.

Following [1], DB is called *stratified* iff we never have both $p \approx q$ and $p \geq_{-1} q$ – that is, within each equivalence class, all dependencies are positive. Following [1], DB is called *strict* iff we never have $p \geq_{+1} q$ and $p \geq_{-1} q$. Let us call DB *semi-strict* iff we never have $p \geq_{-1} p$. It is easy to verify that semi-strict means that all dependencies are strict within each equivalence class – that is,

2.1 Lemma. DB is semi-strict iff for no p and q do we have $p \approx q$, $p \geq_{+1} q$, and $p \geq_{-1} q$. $\square$

Clearly, then, semi-strict follows from either strict or from stratified. We remark briefly on how this will be used in the semantics. *Every* CDB is consistent from the point of view of 3-valued logic, but semi-strict guarantees that we have consistency in 2-valued logic – i.e., that the CDB has a 2-valued model. Strictness is needed to get the stronger fact that every query which is true in all 2-valued models for the CDB is also true in all 3-valued models. Stratifiability is largely irrelevant in this paper, except that it implies semi-strictness, and our construction of 2-valued models in the semi-strict case by induction on $<$ is a generalization of the construction of Apt-Blair-Walker [1] for stratified DB. Semi-strictness was considered also by T. Sato [14] under the name "call-consistency".

If $\mathcal{P} \subseteq PRED$, a *signing* for $\mathcal{P}$ is a map, $S : \mathcal{P} \to \{+1, -1\}$ such that whenever $p, q \in \mathcal{P}$ and $p \leq_i q$, $S(p) = S(q) \cdot i$. Clearly,

2.2 Lemma. If $PRED$ has a signing, then DB is strict. If DB is strict, $\mathcal{P} \subseteq PRED$, a is a $\geq$-largest element of $\mathcal{P}$, and $i \in \{+1, -1\}$, then $\mathcal{P}$ has a signing S defined by: $S(p) = i \cdot k$ whenever $p \in \mathcal{P}$ and $p \leq_k a$. $\square$

As we shall see in §3, signings will allow us to convert 3-valued models to 2-valued models. Whenever a predicate has value **u**, we convert the value to **t** if the sign of the predicate is $+1$, and to **f** if the sign is -1. Strictness does not always imply the existence of a signing. For a counter-example, consider:

a :- p.
a :- q.
b :- p.
b :- $\neg\, q$.

This is strict (and also stratified), but the first 2 clauses prevent p, q from having different signs, while the last 2 clauses prevent them from having the same sign.

The following example illustrates how some of these concepts fit together:

1. a :- $\neg\, b$.
2. b :- $\neg\, a$.
3. a :- p.
4. a :- $\neg\, p$.
5. p :- p.

The equivalence classes are $\{p\}$ at the bottom, with $\{a, b\}$ above it, along with all the trivial singleton equivalence classes of letters not mentioned at all. This database is semi-strict but is neither strict (since $a \geq_{+1} p$ and $a \geq_{-1} p$) nor stratified (since $a \approx b$ and $a \geq_{-1} b$). We can build a 2-valued model for CDB by induction on $<$. Starting at the bottom, we satisfy the completed definition of p; since this is a tautology, $p \leftrightarrow p$, we may assign p either **t** or **f**. We then move up to the equivalence class $\{a, b\}$, and find values for a and b, which satisfy $b \leftrightarrow \neg a$ and $a \leftrightarrow (\neg b \vee p \vee \neg p)$, which means that a must be **t** and b must be **f**; in this example, the value for p turns out to be irrelevant, but if we deleted clauses (1,4), then a would have to have the same value as p. Of course, proposition letters not mentioned at all in the database must be assigned value **f**. So, the CDB for $(1, \ldots, 5)$ is consistent in 2-valued logic. However, a must be true in all 2-valued models, whereas a is not a 3-valued consequence of CDB (and of course, Prolog will not return an answer to the query $?- \ a$). The relevant thing here is that strictness is violated; a depends on p both positively and negatively, which makes it impossible to convert the 3-valued model that assigns **u** to a, b, and p, to a 2-valued model which assigns **f** to a. The lack of stratifiability is irrelevant – if we deleted the (1,2), the DB would become stratified, but we still would have the same problem with a. However, if we deleted (3), the DB would become strict, and we would have a signing defined by $S(a) = -1$; $S(b) = S(p) = +1$. This would allow us to convert the **u** to **f** for a while converting b and p from **u** to **t**.

For queries consisting of more than one literal, we need an addition to our definition of strictness. Consider the database

a :- p.
b :- $\neg$ p.
p :- p.

This is strict (and also stratified). Since $\neg(a \wedge b)$ is a 2-valued consequence of CDB, an answer of "no" would be supported in 2-valued logic to $?- \ a, b$, but no answer is supported in 3-valued logic. The problem is that even though DB is strict, the query clause depends both positively and negatively on p. If ϕ is a query clause, we say $\phi \geq_i p$ iff either $a \geq_i p$ for some a occurring positively in ϕ or $a \geq_{-i} p$ for some a occurring negatively in ϕ. We call *DB strict with respect to* ϕ iff for no predicate letter p do we have both $\phi \geq_{+1} p$ and $\phi \geq_{-1} p$. *DB* could be strict with respect to ϕ but not strict, since strictness could be violated by letters upon which ϕ does not depend. To prove that a given 2-valued consequence, ϕ, of CDB is also a 3-valued consequence, what we shall really need is that DB is strict with respect to ϕ (to allow conversion of 3-valued to 2-valued models for the letters upon which ϕ depends), plus that DB is semi-strict (to allow these 2-valued models to be expanded to models for all the letters).

For the completeness result (with respect to 3-valued logic) in §4, strictness and related notions are irrelevant – the hypothesis there will be *allowedness* [15, 1]. DB is *allowed* iff for each clause,

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

in DB, and each variable, X, which occurs anywhere in that clause, X occurs in at least one *positive* literal, λ_i, in that clause. Likewise, a query clause is called *allowed* iff every variable which occurs in the clause occurs somewhere in a positive literal in that clause. So, for example, the query clause $?- \ \neg q(X), r(X)$ is allowed. In the completeness proof, SLDNF will try to first investigate the positive literal, $r(X)$ in an attempt to bind X and avoid floundering. The fact that DB is allowed guarantees that all intermediate goal clauses are allowed also. Note that the only allowed unit database clauses are ground.

§3. SEMANTICS.

Recall that we are always working with a fixed language in predicate logic. A *3-valued structure*, $\mathcal{A}$, for the language consists of a non-empty set, A (the domain of discourse), together with an appropriate assignment of a semantic object on A for each of the predicate and function symbols of the language. More precisely, whenever

f is an n-place function symbol with $n \geq 1$, $\mathcal{A}(f)$ is a function from A^n into A; if $n = 0$ (f is a constant symbol) then $\mathcal{A}(f) \in A$. Whenever p is an n-place predicate symbol other than '=' with $n \geq 1$, $\mathcal{A}(p)$ is a function from A^n into $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$; if $n = 0$ (p is a proposition letter) then $\mathcal{A}(p) \in \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$. $\mathcal{A}(=)$ is always true identity; i.e., $\mathcal{A}(=)(a, b)$ is $\mathbf{t}$ iff a and b are the same object, and $\mathbf{f}$ otherwise. A *2-valued structure* is simply a 3-valued structure in which the value $\mathbf{u}$ is never taken.

We use Kleene's truth tables of the propositional connectives other than '$\leftrightarrow$', which is given Łukasiewicz's truth table – that is $\mathbf{v} \leftrightarrow \mathbf{w}$ is $\mathbf{t}$ iff $\mathbf{v} = \mathbf{w}$ and $\mathbf{f}$ otherwise. In our applications, the connective $\leftrightarrow$ will occur only as the "iff" in the CDB.

If τ is a term with variables among $X_1, \ldots, X_n$ ($n \geq 1$), then we define, in the obvious way, $\mathcal{A}(\tau) : A^n \rightarrow A$. There is some danger of confusion here, since we do not require that all the X_i actually occur in τ, but this will always be clear from context – to avoid confusion in abstract discussions, we might display τ as $\tau(X_1, \ldots, X_n)$. If τ is a closed term, $\mathcal{A}(\tau) \in A$. Likewise, if ϕ is a formula with free variables among $X_1, \ldots, X_n$ ($n \geq 1$), then we define, in the obvious way, $\mathcal{A}(\phi) : A^n \rightarrow \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$, using the truth tables as in [8]. If ϕ is a sentence (no free variables), then $\mathcal{A}(\phi) \in \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$.

Observe that the interpretation of '=' is always standard 2-valued identity and function symbols are interpreted as functions in the standard mathematical sense. Thus, the truth value, $\mathbf{u}$ is never taken by a formula which is made up of just '=' and function symbols.

Let CET, Clark's Equational Theory, be the equational axioms of the CDB; CET doesn't depend on DB. As in [8], we only consider structures which satisfy CET. These are characterized by the following three conditions. First, $\mathcal{A}(f)$ is a 1-1 function for each function symbol, f of 1 or more places. Second, $\mathcal{A}(f)$ and $\mathcal{A}(g)$ have disjoint ranges whenever f, g are distinct function symbols of 0 or more places. Third, whenever X_i actually *occurs* in the term $\tau(X_1, \ldots, X_n)$, and $a_1, \ldots, a_n \in A$, we have $a_i \neq \mathcal{A}(\tau)(a_1, \ldots, a_n)$; this is a formalization of the *occurs check.*

An *Herbrand* model is a model whose domain of discourse is the set of all closed terms, with function and constant symbols having their natural interpretations. Any model of CET contains an isomorphic copy of an Herbrand model.

We say that $\mathcal{A}$ is a model of the CDB iff all the sentences of the CDB have truth value $\mathbf{t}$ in $\mathcal{A}$. In 3-valued logic, the CDB is always consistent (i.e., has a model). This fact is discussed explicitly in [8], but the result is actually implicit in the earlier paper of Fitting [6]. To construct a 3-valued model of the CDB, first fix a model of CET, and then form a sequence of structures by iterating the 3-valued analogue of the van Emden - Kowalski [5] operator T (details below). All the structures in the sequence have the same domain of discourse and interpretation of the function symbols, but the predicates change; the initial model in the sequence has all predicates undefined ($\mathbf{u}$). The operator T is not in general continuous, so the sequence may be transfinite, but since T is monotone, there is some closure ordinal, at which stage one gets a model for the CDB. Unlike the van Emden - Kowalski case in pure Horn logic, this model is not minimal – that is, it may make some statements true which do not in fact follow from the CDB. For example, if the DB is

```
p(0).
p(s(X)) :- p(X).
```

```
r(0).
r(s(X)) :- r(X).
q :- p(X), ¬ r(X).
```

then q will become $\mathbf{f}$ at stage ω if we start on the Herbrand universe. However, if our original model of CET contained an element which was in the range of s^k for each k, then q will be $\mathbf{u}$ in the final model of the CDB; by our procedure below (Theorem 3.6), this model could then be transformed into a 2-valued model of the CDB in which q is true.

[6] starts with the Herbrand universe. [8] shows essentially that if one begins with an $\aleph_1$-saturated model of CET, then the procedure actually closes at stage ω, and only sentences which follow from the CDB are made true.

We now describe the details of this iteration; in fact, we shall describe a generalization, which is related to the remark make in the introduction that one may define models by recursion on the data dependency relation, $>$. Suppose we have a set of predicate letters, $\mathcal{P} \subseteq PRED$. A *$\mathcal{P}$-structure* is a structure $\mathcal{A}$ as above, except that $\mathcal{A}(p)$ is defined only for $p \in \mathcal{P}$ (but $\mathcal{A}(f)$ is defined for all function symbols). CDB$|\mathcal{P}$ denotes the equational theory, CET, together with all sentences of CDB which only use predicate symbols in $\mathcal{P}$. If $\mathcal{A}$ is a $\mathcal{P}$-structure, $\mathcal{P} \subseteq \mathcal{Q} \subseteq PRED$, and $\mathcal{B}$ is a $\mathcal{Q}$-structure, we call $\mathcal{B}$ an *expansion* of $\mathcal{A}$ iff $\mathcal{B}$ has the same domain of discourse as does $\mathcal{A}$ and agrees with $\mathcal{A}$ on the interpretations of all function symbols and all predicate symbols in $\mathcal{P}$.

If $\mathcal{P}$ is downward closed (see §2), $\mathcal{A}$ is a $\mathcal{P}$-structure, and $\mathcal{A}$ is a model of CDB$|\mathcal{P}$, we shall construct an expansion of $\mathcal{A}$ to a $PRED$-structure which is a model of all of the CDB. Consider DB, $\mathcal{P}$, and $\mathcal{A}$ to be fixed. Let $\mathcal{L}$ be the set of all expansions of $\mathcal{A}$ to $PRED$-structures. We consider $\{\mathbf{t},\mathbf{f},\mathbf{u}\}$ to be a semilattice, whose only relations are $\mathbf{u} < \mathbf{t}$ and $\mathbf{u} < \mathbf{f}$. This makes $\mathcal{L}$ into a complete semilattice in the obvious way. $\mathcal{L}$ has as a bottom element the structure $\mathcal{B}_0$, in which all predicates not in $\mathcal{P}$ are identically $\mathbf{u}$. We define an operator, $T : \mathcal{L} \to \mathcal{L}$ as follows. Given $\mathcal{B} \in \mathcal{L}$, an n-place $p \in PRED$, and $a_1, \ldots, a_n$, we must explain how to compute $\mathbf{v} = T(\mathcal{B})(p)(a_1, \ldots, a_n) \in \{\mathbf{t},\mathbf{f},\mathbf{u}\}$. $\mathbf{v}$ will be $\mathbf{t}$ iff for *some* clause ϕ in DB of form

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

with variables $Y_1, \ldots, Y_m$, and *some* $b_1, \ldots, b_m \in A$, we have $\mathcal{B}(\psi)(b_1, \ldots, b_m) = \mathbf{t}$ and each $\mathcal{B}(\tau_i)(b_1, \ldots, b_m) = a_i$. $\mathbf{v}$ will be $\mathbf{f}$ iff for *every* ϕ in DB of that form, and *every* $b_1, \ldots, b_m \in A$, we have either $\mathcal{B}(\psi)(b_1, \ldots, b_m) = \mathbf{f}$ or some $\mathcal{B}(\tau_i)(b_1, \ldots, b_m) \neq a_i$. $\mathbf{v}$ is $\mathbf{u}$ otherwise. Using the facts that $\mathcal{P}$ is downward closed and $\mathcal{A}$ satisfies CDB$|\mathcal{P}$, one can show that if $p \in \mathcal{P}$, then $T(\mathcal{B})(p) = \mathcal{B}(p) = \mathcal{A}(p)$ – i.e., $T(\mathcal{B})$ indeed is in $\mathcal{L}$. Likewise,

3.1 Lemma. If $\mathcal{B} \in \mathcal{L}$, then $T(\mathcal{B}) = \mathcal{B}$ iff $\mathcal{B}$ is a model for the CDB. □

Since T is monotone, it has a fixed point by the usual Knaster-Tarski Theorem. Hence,

3.2 Theorem. If $\mathcal{P} \subseteq PRED$, $\mathcal{A}$ is a $\mathcal{P}$-structure, $\mathcal{P}$ is downward closed, then there is an expansion of $\mathcal{A}$ which is a (3-valued) model for the CDB. □

Now, suppose we want a 2-valued model for the CDB. Of course, there need not be any, as the well-known example, $p :- \neg p$ shows, but under certain conditions on DB, we may convert 3-valued models to 2-valued models. The simplest condition is the existence of a signing (see §2), which can tell us which **u** values to change to **t** and which to **f**. Actually, the signing need only be defined on predicate letters whose interpretation is **u** at some places. Suppose that $\mathcal{Q}$ and $\mathcal{P}$ are disjoint subsets of $PRED$ and $\mathcal{A}$ is a 3-valued $\mathcal{Q} \cup \mathcal{P}$-structure. Suppose that $S : \mathcal{Q} \to \{+1, -1\}$. Define $2val(\mathcal{A}, S)$ to be the $\mathcal{Q} \cup \mathcal{P}$-structure which has the same domain of discourse as does $\mathcal{A}$, which interprets predicates in $\mathcal{P}$ and function symbols the same way $\mathcal{A}$ does, and in which, for $q \in \mathcal{Q}$, $\mathbf{v} = 2val(\mathcal{A}, S)(q)(a_1, \ldots, a_n)$ is computed as follows: Set $\mathbf{w} = \mathcal{A}(q)(a_1, \ldots, a_n)$; then $\mathbf{v}$ is $\mathbf{w}$ if $\mathbf{w}$ is **t** or **f**; if $\mathbf{w}$ is **u**, then $\mathbf{v}$ is **t** if $S(q) = +1$ and **f** if $S(q) = -1$. One easily verifies:

3.3 Theorem. Suppose that $\mathcal{Q}$ and $\mathcal{P}$ are disjoint subsets of $PRED$, $\mathcal{Q} \cup \mathcal{P}$ is downward closed, and $\mathcal{A}$ is a 3-valued model for CDB$|(\mathcal{Q} \cup \mathcal{P})$ such that the interpretation of each letter in $\mathcal{P}$ happens to be 2-valued. If S is a signing for $\mathcal{Q}$, then $2val(\mathcal{A}, S)$ is a 2-valued model for CDB$|(\mathcal{Q} \cup \mathcal{P})$. □

It follows, of course, that if there is a signing for $PRED$, then the CDB has a 2-valued model, but in fact we can prove this just using the weaker fact that DB is semi-strict. It is here that we use a recursion on $<$ in direct analogy with Apt-Blair-Walker [1].

3.4 Theorem. Suppose that $\mathcal{P}$ and $\mathcal{Q}$ are disjoint subsets of $PRED$, and both $\mathcal{P}$ and $\mathcal{P} \cup \mathcal{Q}$ are downward closed. Suppose also that DB is semi-strict on $\mathcal{Q}$ and that $\mathcal{A}$ is a 2-valued model for CDB$|\mathcal{P}$. Then $\mathcal{A}$ has an expansion to a 2-valued model for CDB$|(\mathcal{P} \cup \mathcal{Q})$.

Proof. Since predicate symbols not mentioned in DB can (must) simply be defined to be universally **f**, we may assume that all symbols in $\mathcal{Q}$ are actually mentioned in DB, so that in particular $\mathcal{Q}$ is finite. Since $<$ is acyclic on $\mathcal{Q}$, we may produce the expansion by recursion on $<$, so that without loss of generality, we may assume that $\mathcal{Q}$ consists of just one equivalence class, which is minimal over $\mathcal{P}$. In this case, there must be a signing, S, for $\mathcal{Q}$. By Theorem 3.2, let $\mathcal{B}$ be an expansion of $\mathcal{A}$ to a 3-valued model for CDB$|(\mathcal{P} \cup \mathcal{Q})$. Then, by Theorem 3.3, $2val(\mathcal{B}, S)$ is a 2-valued model for CDB$|(\mathcal{P} \cup \mathcal{Q})$. □

In particular, taking $\mathcal{P}$ to be empty and $\mathcal{Q}$ to be $PRED$, we have

3.5 Corollary. If DB is semi-strict, then the CDB has a 2-valued model. □

However, as pointed out in §2, semi-strictness is not enough to prove the stronger result that every 2-valued consequence is also a 3-valued consequence. In general, call a formula, ϕ, a *3-valued consequence* of the CDB iff its universal closure, $\forall\phi$, has value **t** in all 3-valued models of the CDB; ϕ is a *2-valued consequence* iff $\forall\phi$ is **t** in all 2-valued models. Obviously, every 3-valued consequence is also a 2-valued consequence.

3.6 Theorem. Suppose that ϕ is a query clause, DB is strict with respect to ϕ, DB is semi-strict, and $\forall\phi$ is a 2-valued consequence of the CDB. Then $\forall\phi$ is also a 3-valued consequence of the CDB.

Proof. Assume that $\mathcal{A}$ is a 3-valued model of the CDB in which $\forall\phi$ has value **f** or **u**. We show how to construct a 2-valued model, $\mathcal{D}$, for the CDB, in which $\forall\phi$ has value **f**. Let $\mathcal{P}$ be the set of all predicate symbols which are $\leq$ some symbol occurring in ϕ. Let S be the (unique) signing for $\mathcal{P}$ such that for each p occurring in ϕ, $S(p)$ is -1 if p occurs positively, and $+1$ if p occurs negatively. Let $\mathcal{B}$ be the reduct of $\mathcal{A}$ to $\mathcal{P}$; that is, $\mathcal{B}$ is exactly the same as $\mathcal{A}$, except that $\mathcal{B}(p)$ is defined only when $p \in \mathcal{P}$. Let $\mathcal{C}$ be $2val(\mathcal{B}, S)$.

$\mathcal{B}$ is a 3-valued, and $\mathcal{C}$ a 2-valued model for CDB$|\mathcal{P}$. In $\mathcal{B}$, ϕ is **u** or **f** at some values of its variables, and at these values, ϕ will be **f** in $\mathcal{C}$, so $\forall\phi$ is **f** in $\mathcal{C}$. By Theorem 3.4, $\mathcal{C}$ has an expansion to a 2-valued model, $\mathcal{D}$, for the CDB, and of course $\forall\phi$ is still **f** in $\mathcal{D}$. $\square$.

Following Clark [3], DB is called *hierarchical* iff it has no recursions whatever; that is, each equivalence class is a singleton, and we never have $p \sqsupseteq p$. For such databases, it is easily proved by induction on $<$ that every 3-valued model for the CDB is in fact 2-valued; it is then immediate that the 3-valued and 2-valued semantics coincide. Clearly, hierarchical implies semi-strict, but it does not imply strict, nor does strict imply hierarchical. One may generalize Theorem 3.6 as follows. Suppose $\mathcal{P} \subseteq PRED$ is downward closed, DB is semi-strict, and DB restricted to $\mathcal{P}$ is hierarchical. Let ϕ be a query clause such that no predicate outside of $\mathcal{P}$ depends both positively and negatively on ϕ. Then $\forall\phi$ is a 2-valued consequence of the CDB iff it is a 3-valued consequence. As special cases, $\mathcal{P} = \emptyset$ is Theorem 3.6, and $\mathcal{P} = PRED$ is the fact that the 2-valued and 3-valued semantics agree for hierarchical databases.

§4. A COMPLETENESS RESULT.

Here we deal only with 3-valued models, and study the concept of allowedness (see §2). If ϕ is a formula, we say that CDB $\models \phi$ iff the universal closure, $\forall\phi$ has value **t** in all 3-valued models of the CDB. Although, as we have pointed out, in general it is necessary to consider models which extend the Herbrand universe, we note that by [8], we may generate consequences of the CDB by a construction which only involves the Herbrand universe. For this section, let $\mathcal{A}_0$ be the structure whose domain of discourse is the Herbrand universe, and in which all predicates are identically **u**. For a given DB, we may iterate T as above and define $\mathcal{A}_{n+1} = T(\mathcal{A}_n)$. Although none of the $\mathcal{A}_n$ nor their limit, $\mathcal{A}_\omega$, need be a model for the CDB, we have:

4.1 Theorem (see [8]). For any ϕ, CDB $\models \phi$ iff $\forall\phi$ is **t** in $\mathcal{A}_n$ for some n. $\square$

If one wants to prove a completeness theorem of the form that every supported answer is eventually returned, it is natural, then, to induct on the stage at which the answer gets supported.

Let Q be the set of all query clauses. Let RET be the set of all pairs, (ϕ,σ), where $\phi \in Q$, σ is a substitution, and σ acts only on the variables occurring in ϕ. If ϕ is ground, the only possibility for σ is "yes", the identity substitution. RET is the set of all possible (query, answer) pairs. If σ is any substitution and $\phi \in Q$, let $\sigma|\phi$ be σ restricted to the variables actually occurring in ϕ; so $(\phi, \sigma|\phi)$ is always in RET.

The semantic import of allowedness is given by:

4.2 Theorem. Suppose DB is allowed and ϕ is an allowed query. For each n, there are only finitely many σ such that $(\phi,\sigma) \in RET$ and $\forall\phi\sigma$ is **t** in $\mathcal{A}_n$; furthermore, all such σ are ground.

Proof. Induction on n. $\square$

The next step is to say precisely what we mean by SLDNF. This is defined in [12], but the definition is a little complicated, as one must simultaneously define what a derivation is and what a finitely failed SLDNF derivation tree is. Since we do not actually need the structure of these derivations, but only the set of derivable answers, it is a little easier to work directly with an inductive definition of the set of answers returnable by SLDNF. We define a subset, $\mathbf{R} \subseteq RET$ (the set of all such pairs actually obtainable by SLDNF with the given database), and a subset $\mathbf{F} \subseteq Q$ (the set of finitely failable queries). $\mathbf{R}$ and $\mathbf{F}$ are defined simultaneously by recursion to be the least sets satisfying the following closure properties:

0 $true\mathbf{R}yes$.

R+ If ϕ is $\alpha \wedge \psi$, α is a positive literal, $(\beta :- \theta) \in$ DB, $\sigma = mgu(\alpha, \beta)$, and $(\theta \wedge \psi)\sigma\mathbf{R}\pi$, then $\phi\mathbf{R}(\sigma\pi)|\phi$.

R− If ϕ is $\neg\alpha \wedge \psi$, α a positive *ground* literal, $\alpha \in \mathbf{F}$, and $\psi\mathbf{R}\sigma$, then $\phi\mathbf{R}\sigma$.

F+ Suppose that ϕ is $\alpha \wedge \psi$ and α is a positive literal. Suppose that for *each* $(\beta :- \theta) \in$ DB, *if* β is unifiable with α, *then* $(\theta \wedge \psi)(mgu(\alpha, \beta)) \in \mathbf{F}$. Then $\phi \in \mathbf{F}$.

F− If ϕ is $\neg\alpha \wedge \psi$, α a positive *ground* literal, and $\alpha\mathbf{R}yes$, then $\phi \in \mathbf{F}$.

In the above, before computing an mgu, we assume that the query clause and the database clause are renamed to have disjoint variables – the usual "standardized apart" assumption. When we say that ϕ is $\lambda \wedge \psi$ (λ is α or $\neg\alpha$), we mean that λ is *some* literal occurring in ϕ, and ψ is ϕ with one occurrence of λ deleted. Note that clause F+ includes as a special case that if α fails to unify with the head of any clause in the database, then $\alpha \wedge \psi$ fails.

It is easy to see that this definition is equivalent to the one via SLDNF trees – in fact, an SLDNF tree headed by a query, ϕ, may be viewed as a backward search for a σ such that $\phi\mathbf{R}\sigma$. We may thus state our completeness result formally as:

4.3 Theorem. Suppose that DB is allowed and ϕ is an allowed query clause. If CDB $\models \phi\sigma$, then $\phi\mathbf{R}\sigma$. If CDB $\models \neg\exists\phi$, then $\phi \in \mathbf{F}$.

The proof will be "by induction". Before beginning, let us note some features that the induction must have. First, since $\mathbf{F}$ and $\mathbf{R}$ are defined simultaneously in terms of each other, we must prove both parts of the theorem simultaneously, presumably by induction on the least n such that the relevant statement is $\mathbf{t}$ in $\mathcal{A}_n$. Second, the induction cannot be quite that simple, since the reduction from n to $n-1$ could take an unspecified number of SLDNF steps. In the positive case, if CDB $\models \phi\sigma$, then σ must be ground, which makes matters a little easier, but suppose ϕ is $p(X), \neg q(X)$; perhaps $\neg q(X)\sigma$ first becomes $\mathbf{t}$ at stage 10, whereas $p(X)\sigma$ is $\mathbf{t}$ by stage 5. We would like to work on the $\neg q(X)$ first, to reduce the 10 to 9, but of course we cannot – we must work on $p(X)$ until the X becomes ground. Thus, we must actually induct on the stages at which the various literals within a clause become $\mathbf{t}$ individually.

Let SEQ be the set of all finite (possibly empty) sequences of natural numbers. We'll exhibit them in the form $\vec{n} = (n_1, \ldots, n_L)$, where L is the length of $\vec{n}$. Let $\leq$ be the least transitive reflexive relation on SEQ such that $\vec{m} \leq \vec{n}$ whenever $\vec{m}$ is a subsequence of a permutation of $\vec{n}$, or $\vec{m}$ results by replacing some component, n_i, of $\vec{n}$, by an arbitrary finite sequence of numbers less than n_i. Let $<$ be the corresponding strict partial order, so $\vec{m} < \vec{n}$ iff $\vec{m} \leq \vec{n}$ and $\vec{n} \not\leq \vec{m}$ – equivalently, iff $\vec{m} \leq \vec{n}$ and $\vec{m}$ is not a permutation of $\vec{n}$. So, for example,

$$() < (0,1,1,2,3) \leq (1,3,1,2,0) < (4,2,0) < (4,2,0,0) \enspace .$$

As is well-known, the relation $<$ is well-founded, so we may induct on it.

We use these sequences to give a ranking to when the various components of a query clause become true. More precisely, say ϕ is the clause $\alpha_1, \ldots, \alpha_L$. If CDB $\models \phi\sigma$, then let $rank^+(\phi\sigma)$ be $(n_1, \ldots, n_L)$, where each n_i is the minimal n such that $\forall\alpha_i$ is $\mathbf{t}$ in $\mathcal{A}_n$. In particular, $rank^+(true) = ()$. The negative case is more complicated. If CDB $\models \neg\exists\phi$, call $(n_1, \ldots, n_L)$ *a negative rank* for ϕ iff for each *ground* instance, $\phi\sigma$ of ϕ, some $\alpha_i\sigma$ is f in $\mathcal{A}_{n_i}$. Since $<$ is well-founded, ϕ must have a *minimal* negative rank, but this minimal rank need not be unique. For example, with the empty database, p, q is $\mathbf{u}$ at stage 0 and f at stage 1, so that (0,1) and (1,0) are both minimal negative ranks for p, q. Nevertheless, we may prove by induction on $\vec{n}$ that, assuming DB is allowed,

H($\vec{n}$): Whenever ϕ is an allowed query clause:

R: If CDB $\models \phi\sigma$ and $rank^+(\phi\sigma) = \vec{n}$, then $\phi\mathbf{R}\sigma$.

F: If CDB $\models \neg\exists\phi$ and $\vec{n}$ is a negative rank for ϕ, then $\phi \in \mathbf{F}$.

Note that a positive rank is non-0 on all its places, whereas a negative rank may be 0 in some places; this will make the negative cases somewhat harder, as we do not always have the same freedom to choose a literal to reduce.

Proof of Theorem 4.3. The basis case for H($\vec{n}$) is with $\vec{n} = ()$ and hence ϕ empty (*true*); this follows trivially from clause 0 in the definition of $\mathbf{R}$. For the induction step, ϕ is non-empty, and there are two things to prove, and each has two cases depending on whether we reduce a positive or a negative literal. These 4 cases correspond in a natural way to the 4 inductive clauses in the definition of $\mathbf{R}$ and $\mathbf{F}$. We are assuming, inductively, that H($\vec{m}$) holds whenever $\vec{m} < \vec{n}$.

R+: If ϕ contains a positive literal, let us assume that ϕ is $\alpha_1 \wedge \psi$, where α_1 is positive and the first literal in ϕ. (since the order of a clause does not matter here). Since ϕ is allowed, σ is ground (Theorem 4.2). Since $\alpha_1\sigma$ is t in $\mathcal{A}_{n_1}$, there must be some clause, $\beta :- \theta$ in DB and a ground substitution, π such that $\sigma = (mgu(\alpha_1, \beta))\pi|\phi$, and $\theta(mgu(\alpha_1, \beta))\pi$ is t in $\mathcal{A}_{n_1 - 1}$. Then

$$rank^+((\theta \wedge \psi)mgu(\alpha_1, \beta)) \leq \vec{m} < \vec{n} \quad ,$$

where $\vec{m}$ is obtained by replacing the n_1 in $\vec{n}$ with a string of $length(\theta)$ copies of $n_1 - 1$. Applying the inductive hypothesis and the R+ clause in the definition of $\mathbf{R}$, we have that $\phi\mathbf{R}\sigma$.

F+: If possible, permute ϕ to the form $\alpha_1 \wedge \psi$, where α_1 is positive and $n_1 > 0$. So, ψ is of the form $\alpha_2, \ldots, \alpha_L$. Fix *any* clause, $\beta :- \theta$ in DB such that β is unifiable with α_1. We must show that $\delta = (\theta \wedge \psi)(mgu(\alpha_1, \beta))$ is in $\mathbf{F}$, and this will follow by the inductive hypothesis if we can show that $\vec{m}$ (defined exactly as above) is a negative rank for δ. Consider any ground instance, $\delta\pi$. If $\theta(mgu(\alpha_1, \beta))\pi$ is f in $\mathcal{A}_{n_1 - 1}$, we are done, so assume it is t or $\mathbf{u}$. Then $\alpha_1(mgu(\alpha_1, \beta))\pi = \beta(mgu(\alpha_1, \beta))\pi$ is t or $\mathbf{u}$ in $\mathcal{A}_{n_1}$, so, since $\vec{n}$ is a negative rank for ϕ, $\alpha_i(mgu(\alpha, \beta))\pi$ is f in $\mathcal{A}_{n_i}$ for some $i \geq 2$. In either case, some literal in $\delta\pi$ becomes false in the corresponding $\mathcal{A}_{m_j}$.

R$-$: If ϕ has no positive literals, it must be ground. Write ϕ as $\neg\alpha_1 \wedge \psi$. Then $rank^+(\psi\sigma) = (n_2, \ldots n_L) < \vec{n}$, so by the inductive hypothesis, $\psi\mathbf{R}yes$. The unit, α_1 is f in $\mathcal{A}_{n_1}$ so (n_1) is a negative rank for it. If $L \geq 2$, then $(n_1) < \vec{n}$, so the inductive hypothesis applies immediately to show $\alpha_1 \in \mathbf{F}$. If $L = 1$, then $(n_1) = \vec{n}$, and note that $\alpha_1 \in \mathbf{F}$ by the case F+, which we just did for $\vec{n}$.

F−: Say ϕ is $\phi_{pos} \wedge \phi_{neg}$, where ϕ_{pos} is $\alpha_1, \ldots, \alpha_p$, ϕ_{neg} is $\neg\beta_1, \ldots, \neg\beta_q$, and the α_i and β_j are all positive. If we cannot reduce this by F+, then $n_i = 0$ for $i = 1, \ldots, p$. This may then be handled analogously to R− if we can show that some β_j is ground and $\mathbf{t}$ in $\mathcal{A}_{n_{p+j}}$. Say $X_1, \ldots, X_r$ are the variables occurring in ϕ. Let H be the set of all ground terms. Since $\vec{n}$ is a negative rank for ϕ and all truth values are $\mathbf{u}$ in $\mathcal{A}_0$, we must have that for each $\gamma_1, \ldots \gamma_r \in H$, some $\neg\beta_j(\gamma_1, \ldots \gamma_r)$ is $\mathbf{f}$ in $\mathcal{A}_{n_{p+j}}$. Let

$$S_j = \{(\gamma_1, \ldots, \gamma_r) \in H^r : \mathcal{A}_{n_{p+j}}(\beta_j)(\gamma_1, \ldots, \gamma_r) = \mathbf{t}\} \quad .$$

We have just seen that $\bigcup_{j=1}^{q} S_j = H^r$. If β_j is not ground, then the projection of S_j on any coordinate corresponding to an X_i which actually occurs in β_j is finite (by Theorem 4.2). If β_j is ground but $\mathbf{f}$ in $\mathcal{A}_{n_{p+j}}$, then S_j is empty. Since H is infinite, it cannot be the union of finitely many sets each of whose projection on some coordinate is finite, so some β_j is ground and $\mathbf{t}$ in $\mathcal{A}_{n_{p+j}}$. □

§5. CONCLUSION.

One defect in the above completeness result is that the concept of allowedness is so stringent as to exclude many common Prolog constructs, such as the definition of equality ($equal(X, X)$), and both clauses in the standard definition of $member(X, L)$. As a topic for future work, one would like to replace allowedness and strictness by weaker conditions; these conditions should be easily testable by a compiler and weak enough to include the kinds of programs that people commonly write.

One might get better completeness results by strengthening SLDNF to compute more answers. For example, perhaps a query, ?− $\neg p(X)$ should not always flounder. Following the lead of IC-Prolog [4], if ?− $p(X)$ returns the identity, *yes*, it is safe to fail ?− $\neg p(X)$, and if ?− $p(X)$ fails, it is safe to return *yes* for ?− $p(X)$. Then, with the empty database, ?− $\neg p(X)$ would return *yes* and ?− $\neg\neg p(X)$ would return *no*.

Another way to get better completeness results would be to weaken the semantics to support fewer answers. Consider the database:

```
p :- ¬q(X).
q(c).
q(X) :- ¬r(X).
r(c).
```

This is both hierarchical and strict, so the 2-valued and 3-valued semantics are equivalent, and they support a *yes* answer to ?− $\neg p$, although nothing resembling Prolog will return this answer (see also [9] for more on this sort of example). Perhaps the semantics should be changed so as not to support this answer. One way to do this would be to change the notion of "model" to allow structures in which '=' is interpreted as any 3-valued relation which makes the equality axioms come out $\mathbf{t}$. This would allow a model to contain an element, a, such that $a = c$ has truth value $\mathbf{u}$. Then $r(a)$, $q(a)$, and p could be $\mathbf{u}$ as well. The problem with this approach is that now failure would not be supported for ?− $q(X), \neg q(X)$, so that SLDNF would not be sound with respect to this semantics.

An alternate approach would be to use intuitionistic logic. In this example, $\neg p$ does not follow from the CDB in standard intuitionistic proof theory. One cannot, however, just use intuitionistic proof theory, since if the database is $p :- p$, an answer of "no" to the query $?- p, \neg p$ should not be supported, whereas $\neg(p \wedge \neg p)$ is provable intuitionistically. Shepherdson [16] has suggested amalgamating intuitionism with 3-valued logic; in this regard, see also Fitting's [7].

§6. REFERENCES.

[1] Apt, K. R., Blair, H. A., and Walker, A., Towards a Theory of Declarative Knowledge, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufman, Los Altos, 1987.

[2] Cavedon, L., and Lloyd, J. W., A Completeness Theorem for SLDNF Resolution, *to appear.*

[3] Clark, K. L., Negation as Failure, in: H. Gallaire and J. Minker (eds.), *Logic and Databases*, Plenum Press, New York, 1978, pp. 293-322.

[4] 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 Press, New York, 1982, pp. 253-266.

[5] Van Emden, M. H. and Kowalski, R., The Semantics of Predicate Logic as a Programming Language, *JACM* 23:733-742 (1976).

[6] Fitting, M., A Kripke-Kleene Semantics for Logic Programs, *J. Logic Programming* 2:295-312 (1985).

[7] Fitting, M, Logic Programming on a Topological Bilattice, *to appear.*

[8] Kunen, K., Negation in Logic Programming, *J. Logic Programming*, to appear.

[9] Kunen, K., Answer Sets and Negation-As-Failure, in: J.-L. Lassez (ed.), *Logic Programming* (Proceedings of the Fourth International Conference), MIT Press, 1987, pp. 219-228.

[10] Lassez, C., McAloon, K., and Port, G., Stratification and Knowledge Base Management, in: J.-L. Lassez (ed.), *Logic Programming* (Proceedings of the Fourth International Conference), MIT Press, 1987, pp. 136-151.

[11] Lassez, J-L., and Maher, M. J., Optimal Fixedpoints of Logic Programs, *Theoretical Computer Science* 39:15-25 (1985).

[12] Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, 1984.

[13] Mycroft, A., Logic Programs and Many-valued Logic, *Proc. of Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science* 166:274-286 (1984).

[14] Sato, T., On consistency of first order logic programs, *to appear.*

[15] Shepherdson, J.C., Negation as Failure, *J. Logic Programming* 1:51-79 (1984).

[16] Shepherdson, J.C., Negation in Logic Programming, *Proc. of Workshop on Foundations of Deductive Databases and Logic Programming*, Washington, D.C., 1986.