SOME REMARKS ON THE COMPLETED DATABASE

Kenneth Kunnen

Computer Sciences Technical Report #775

JUNE 1988

SOME REMARKS ON THE COMPLETED DATABASE[1]

Kenneth Kunen[2]

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

May 31, 1988

ABSTRACT

We show how Clark's Completed Database (CDB) can be extended to include a number of Prolog constructs usually considered to be outside of pure logic, while still maintaining the known partial completeness results for SLDNF derivations. We also prove that in the semantics provided by the CDB and SLDNF, there is no definition of transitive closure which is strict and does not use function symbols.

§1. **INTRODUCTION.** We begin by reviewing briefly the current status of negation in logic programming. A *general program clause* is a clause whose head is a positive literal and whose body consists of 0 or more positive or negative literals. A *general program* is a finite set of general program clauses. A *positive* program is one in which all literals in the bodies of all clauses are positive. By *declarative semantics*, we mean that the semantics is defined in terms of standard logical notions, such as *model* and *logical consequence*. *Procedural semantics* refers to the actual procedure by which queries are evaluated. As Clark pointed out, we must realize that the program asserts more than its naive interpretation would indicate. Given a program, P, he formed its *completion*, $Comp(P)$, which consists of some equality axioms plus a completed definition of each predicate symbol; roughly, these completed definitions are obtained by replacing the Prolog "if" by an "iff"; see, e.g., [8, 13, 17] for a precise definition. As is by now well known ([11,13,14,16,18]), in order to capture the intended meaning of arbitrary general programs, it is necessary to consider models within 3-valued logic, rather than 2-valued logic. This logic has three truth values, **t** (true), **f** (false), and **u** (undefined). Following Kleene, the truth value **u** corresponds to the notion of a computation which fails to halt. Since 3-valued logic is somewhat unpleasant to deal with, there has been much investigation recently into finding sufficient conditions on programs which allow one to use 2-valued models after all.

Within the framework of the above generalities, there are presently two competing approaches to the semantics of logic programs. One, which we shall call the *logical consequence approach*, says that a query is supported by the program, P, iff it is a logical consequence of $Comp(P)$ – i.e., true in *all* models of $Comp(P)$. The other, which we shall

[1] A shortened version of this paper will be presented at the *Fifth International Conference Symposium on Logic Programming*, Seattle, 1988

[2] This research was supported by NSF Grant DMS-8501521.

call the *canonical model approach*, picks out one specific model of $Comp(P)$, and says that a query is supported iff it is true in that one model.

Regarding the **logical consequence approach**, we show in [13] how to build a mathematically coherent semantics by considering the logical consequences of $Comp(P)$ within 3-valued logic. Then, in [14], we show that if P satisfies a *strictness* condition (see also §2), then the 3-valued and 2-valued logical consequences of P are the same. If we take this approach to the declarative semantics, then the appropriate procedural semantics is given by the notion of an SLDNF derivation [17], which is essentially a formalization of what Prolog does (actually, would do if it used a non-deterministic evaluation order). As is well-known, SLNDF is sound but not complete; that is, there are very simple examples where a query follows logically from $Comp(P)$, but is not supported by SLDNF. However, [7,14] show certain sufficient conditions under which completeness holds. The approach of [14] shows that if we take the 3-valued semantics as the "official" one, then the only condition needed for completeness is *allowedness* (see §2); one can, then, add the strictness condition if one wants completeness with respect to 2-valued logic. The completeness results in [7,14] are stated with respect to pure logic. However, Prolog contains many features which seem to go beyond pure logic. Some of these can be accounted for by appropriate modifications to the definitions of $Comp(P)$ and SLDNF; we discuss this further in §4. In most cases, the needed modifications to SLDNF have already been discussed by Naish [19] and implemented in NU-Prolog [26], but one must investigate how these modifications affect completeness and soundness results. In the case of the arithmetic and term comparison predicates, we can show (Theorem 4.2) that completeness is maintained by an appropriate extension in the definition of "allowed".

Regarding the **canonical model approach**, the problem is, of course, to pick out the one specific model in a natural way. It is always natural to take the Herbrand universe as the domain of discourse. The most general method of choosing a model is Fitting's [11], which, for an arbitrary P, gives a direct construction of a 3-valued Herbrand model of $Comp(P)$. However, here too, there is a preference for obtaining 2-valued models whenever possible. In particular, if the program happens to be stratified, Apt-Blair-Walker [2] and Van Gelder [27] present a natural construction of a 2-valued Herbrand model. Following [2], we shall call this model the *standard interpretation.* In the case of positive programs, the standard interpretation is precisely the van Emden - Kowalski [10] minimal model. The procedural semantics appropriate to the standard interpretation is bottom-up evaluation, as defined in [10] for positive programs and in [2, 27] for stratified programs. If there are no function symbols, this evaluation procedure is amenable to many optimizations, both for positive programs (see [6] for a survey and further references), and for general stratified programs (see [5]). Especially in database applications, it is now common to take this standard interpretation as the canonical model for stratified programs; that is, one says that a query, ϕ, is *supported* iff ϕ is true in the standard interpretation. For example, [4,15] discuss knowledge base management in this framework.

These approaches to semantics are incompatible; for example, Przymusinski [22] points out that standard definition of transitive closure, which optimizes (see [6]) to a very efficient computation in the bottom-up setting, is not even correct in the logical consequence setting; we prove (Theorem 3.1) that in this setting, there is no correct definition at all which is strict and does not use function symbols.

It is hard to say which approach is "better". Certainly, the logical consequence approach provides a better explanation of Prolog, but the problems with transitive closure might lead one to suggest that Prolog be abandoned in favor of bottom-up evaluation. On the other hand, if one wishes to use terms to represent data structures, then Prolog seems preferable, since at least it corresponds to *some* evaluation procedure (SLDNF). In the presence of function symbols, bottom-up evaluation, which involves a search through the (infinite) Herbrand universe, is not an implementable procedure; in the worst case, the standard interpretation is not even recursively enumerable, so that it does not correspond to any evaluation procedure at all [3]. In the logical consequence semantics of [13], the set of supported queries is always recursively enumerable, but one pays for this by requiring the theory to consider non-Herbrand models. There are examples of queries which are true in all Herbrand models (3-valued as well as 2-valued) of $comp(P)$, but not in all models, and thus are not supported by the logical consequence semantics. The examples in [13, 14] seem somewhat artificial, but when we add in arithmetic predicates in §4, many natural examples arise from the fact that computations cannot search through all numbers.

§**2. SYNTACTIC NOTATION.** We assume that our language for predicate logic is fixed in advance, 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 program, but is used in forming $Comp(P)$.

We follow the notation in [14] for describing data dependency relations, although these notions, using different notation, occur much earlier. 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 P 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 either p is q or p hereditarily depends on q.

A *signed* dependency is defined as follows. We say $p \sqsupseteq_{+1} q$ iff there is a clause in P 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 P 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

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

So, $p \geq q$ implies that either $p \geq_{+1} q$ or $p \geq_{-1} q$ (or both).

Following [2], P is called *stratified* iff we never have both $p \geq_{-1} q$ and $q \geq p$ – that is, there are no dependency cycles through a negation. Following [2], P is called *strict* iff we never have $p \geq_{+1} q$ and $p \geq_{-1} q$. Following Sato [23], P is called *call-consistent* iff we never have $p \geq_{-1} p$. Since we always have $p \geq_{+1} p$, call-consistency follows from either strictness or from stratifiability.

For queries consisting of more than one literal, we need an addition to our definition of strictness, which says, approximately, that we are considering the entire query to be a new predicate. If ϕ is a query clause, we say $\phi \geq_i p$ iff either $q \geq_i p$ for some q occurring positively in ϕ or $q \geq_{-i} p$ for some q occurring negatively in ϕ. We call P *strict with respect to* ϕ iff for no predicate letter p do we have both $\phi \geq_{+1} p$ and $\phi \geq_{-1} p$. P could be strict with respect to ϕ but not strict, since strictness could be violated by letters upon which ϕ does not depend. The actual strictness condition used in [14] to conclude that a given 2-valued consequence, ϕ, of $Comp(P)$, is also a 3-valued consequence, was that P be strict with respect to ϕ and that the whole P be call-consistent.

For the completeness result (with respect to 3-valued logic) in [14], what was required was that both the program and the query be *allowed* [24, 2]. P is *allowed* iff for each clause, $\alpha :- \lambda_1, \ldots \lambda_n$, in P, 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.

§3. MODELS.

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 assignment of an appropriate semantic object on A for each of the predicate and function symbols of the language. More precisely 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 $\mathcal{A}(p) \in \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$. In our models, 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.

We use Kleene's truth tables of the propositional connectives other than '$\Longleftrightarrow$', which is given Łukasiewicz's truth table – that is $\mathbf{v} \Longleftrightarrow \mathbf{w}$ is $\mathbf{t}$ iff $\mathbf{v} = \mathbf{w}$ and $\mathbf{f}$ otherwise. In our applications, the connective $\Longleftrightarrow$ will occur only as the "iff" in $Comp(P)$. A more detailed discussion of the choice of truth tables is given in [13].

If S is a set of sentences, a *3-valued model* of S is just a 3-valued structure in which all sentences of S have truth value $\mathbf{t}$. A sentence ϕ is a 3-valued consequence of S iff ϕ has value $\mathbf{t}$ in all 3-valued models of S. A *2-valued structure* is simply a 3-valued structure in which the value $\mathbf{u}$ is never taken. Thus, any 3-valued consequence of S is also a logical consequence of S in the standard, 2-valued sense – namely, true in all 2-valued models of S. In the i-valued logical consequence approaches (where i is 2 or 3), a query, ϕ, is said to be *supported* by P iff ϕ is an i-valued consequence of $Comp(P)$; equivalently, iff ϕ has truth value $\mathbf{t}$ in all i-valued models of $Comp(P)$. Thus, if ϕ is supported under the 3-valued semantics, it is also supported under the 2-valued semantics, and the converse is true as well under the strictness condition. Note that the semantics is defined without any regard to possible procedures for constructing models; in proving theorems, as in [11,13,14], it is often useful to build models by iterating a 3-valued analogue of the van Emden - Kowalski [10] T_P operator.

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. Because of the equality axioms, any model of $Comp(P)$ contains an isomorphic copy of an Herbrand model; but it may properly extend the Herbrand universe – that is, it may contain objects which are not the denotations of any closed terms. As is well-known (see, e.g., [13, 14]), it is in general necessary to consider non-Herbrand models; that is, there can be a ϕ which is true in all Herbrand models of $Comp(P)$ but not true in all models; then, of course, by soundness, ϕ will not be supported by SLDNF; in fact, this can even happen with positive programs if we allow the query to be negative.

In the case of positive programs with *positive* queries, one *can* restrict oneself to Herbrand models, and furthermore, iterating the original (2-valued) T_P produces a *minimal* model of $Comp(P)$; that is, any positive query which is not supported is already **f** in this minimal model. In this case, then, we have two equivalent procedures for evaluating queries: top-down (SLD), or bottom-up (iterating T).

However, as soon as we allow *negative* queries to our positive program, the two approaches lead to incompatible views of negation [2, 25]. Roughly, what has happened here is that the truth values **u** and **f** in the 3-valued $Comp(P)$ semantics both got identified to **f** in the minimal model. In the 2-valued approach, we have our truth values ordered $\mathbf{f} < \mathbf{t}$, which has the effect of making **f** the default value. In the 3-valued approach, our truth values have $\mathbf{u} < \mathbf{t}$ and $\mathbf{u} < \mathbf{f}$, with no relationship between **t** and **f**; this has the effect of making **u** the default value, and creating a symmetry between the values **t** and **f**. For a trivial example, suppose that P consists of the one clause, $p :- \ p$. Then $Comp(P)$ is $p \iff p$ and supports neither p nor $\neg p$, whereas $\neg p$ is **t** in the minimal model and is supported by bottom-up evaluation.

When we add negation into the program, there is not in general any way to choose a canonical model in 2-valued logic. For example, the completion of $\{p :- \ \neg p\}$ has no 2-valued models at all. Even when the strictness conditions are met (see §2), so that we can replace 3-valued by 2-valued models, there is no natural choice of a 2-valued model. For example, if P is $\{p :- \ \neg q\ ,\ \ q :- \ \neg p\}$, 2-valued models of $Comp(P)$ must have exactly one of p, q being **t**, and there is no natural way to choose which one. Of course, with 3-valued models, the minimal (and natural) model makes both p and q **u**.

In the case of stratified programs, however, there is a natural construction of a 2-valued model by induction on the levels of the stratification [2, 27]; following [2], we refer to this model as the *standard interpretation*. These methods extend to provide natural 2-valued models for certain non-stratified programs as well [21].

Since the standard interpretation reduces to the minimal model in the case of positive programs, the trivial program, $\{p :- \ p\}$, already points out the difference between this approach and the logical consequence one. For a more interesting example, consider the problem of defining a 2-place predicate, tc, to be the transitive closure of another 2-place predicate, p. Of course, transitive closure is not first-order definable in general, but we have in mind a more modest goal. Let us call a *fact about* p any ground unit clause with p as its predicate symbol. If ρ is any finite relation on (i.e. set of ordered pairs from) the Herbrand universe, let F_ρ be the finite set of facts which encode it into p:

$$F_\rho = \{p(\alpha, \beta) : \alpha\rho\beta\} \ .$$

Let ρ^* be the transitive closure of ρ; this is also finite. Let D be any finite set of clauses not having p as their head predicate symbol. We say that D *defines tc* to be the transitive closure of p with respect to a given semantics iff for every such ρ and all closed terms, α and β: $tc(\alpha, \beta)$ is supported by the program $D \cup F_\rho$ if $\alpha\rho^*\beta$, and $\neg tc(\alpha, \beta)$ is supported otherwise.

The most obvious solution is to take D to be:

$$tc(X, Y) :- \; p(X, Y).$$
$$tc(X, Z) :- \; p(X, Y), tc(Y, Z).$$

If we use the standard interpretation as our semantics, D does indeed define tc to be the transitive closure of p in the above sense. In fact examples like this one are commonly used to illustrate the success of query optimization methods [6] with bottom-up evaluation in this setting.

However, as is pointed out in [22], D does not correctly define transitive closure with respect to the logical consequence semantics. For example, say $F_\rho = \{p(a, b), p(b, a), p(c, a)\}$, and we query the system $?- \; tc(a, c)$. Procedurally, we get an infinite loop. Declaratively, neither $tc(a, c)$ nor its negation follows from $Comp(D \cup F_\rho)$. The obvious model of $Comp(D \cup F_\rho)$ has $tc(a, c)$ false, but there is another model in which tc is true of all pairs from $\{a, b, c\}$.

One might hope that since this program is correct (both procedurally and declaratively) in the case that ρ happens to be acyclic, some minor modification of it will work for a general ρ, but this is not true.

3.1. Theorem. Suppose D is strict and does not use any function symbols (except for constants). Then D cannot define tc to be the transitive closure of p with respect to the logical consequence semantics.

We defer the proof until the end of this section. The proof makes use of the fact, from model theory, that the transitive closure of a relation is not *explicitly* definable by a first-order formula (see [1]). However, we cannot simply quote this fact, since the program might provide an *implicit* definition, perhaps making use of auxiliary predicates. In fact, the following three remarks show that minor variants of the theorem are false, indicating that it is unlikely that the theorem could be deduced in a really trivial way from general model-theoretic facts.

(1) If we allow function symbols, then it is possible to write a correct definition of transitive closure which is both strict and stratified. For example, one may use the standard Prolog definition of *member*, together with the following, borrowed from §7.2 of Clocksin-Mellish [9]:

$$tc(X, Y) :- \; go(X, Y, [])$$
$$go(X, Y, L) :- \; \neg member(X, L), p(X, Y), \neg member(Y, L)$$
$$go(X, Y, L) :- \; \neg member(X, L), p(X, Z), go(Z, Y, [X|L])$$
$$go(X, X, L) :- \; \neg member(X, L), p(X, Z), go(Z, X, L)$$

The auxiliary predicate $go(X, Y, L)$ "says" that L is a list and there is a path from X to Y whose intermediate nodes do not contain any members of L. The correctness of $go(X, Y, L)$, when called with X, Y, L ground, can be proved by induction on the number of nodes not listed in L.

(2) Even if we ban function symbols, there are properties which are uniformly definable from p by strict (in fact, positive) programs under the logical consequence semantics which are not first-order definable. For example, one can define "has a cycle" by: $cyclic :- \ tc(X, X)$, plus the two clauses above in the definition of tc. That is, if the program, P, consists of these three clauses, together with a finite number of ground facts about p (representing some finite relation ρ), then either:

a. ρ has a cycle, in which case in every 3-valued model of $comp(P)$, $cyclic$ has truth value $\mathbf{t}$ (since $tc(X, X)$ is $\mathbf{t}$ at some value of X), or
b. ρ is acyclic, in which case in every 3-valued model of $comp(P)$, tc has it's expected (2-valued) interpretation, so cyclic is $\mathbf{f}$.

However, "has a cycle" is not uniformly first-order definable from p.

(3) If one adds in @< (lexical order), as described in §4, then one can define tc by the positive program:

$$
\begin{array}{l}
tc(X, Y) :- \ q(X, Y, A, B). \\
q(X, Y, A, B) :- \ p(X, Y), node(A), node(B), A@\!<\, B. \\
q(X, Y, A, B) :- \\
\qquad p(X, Z), node(A), node(B), node(C), A@\!<\, C, C@\!<\, B, \ q(Z, Y, C, B). \\
node(A) :- \ p(A, B). \\
node(A) :- \ p(B, A).
\end{array}
$$

Here, $q(X, Y, A, B)$ "says" that $A@\!<\, B$ and there is a p-path from X to Y of length no longer than the number of @< steps from A to B.

In (1) and (3), the programs actually work (very slowly!) in standard Prolog. Likewise in (2) if the graph is acyclic; if the graph has a cycle, the program would have to list the cycle first in the definition of p for the standard (left-right) evaluation of Prolog to terminate.

In light of these remarks, it would be interesting to know whether there is a generalization of Theorem 3.1 which is not specific to the special case of transitive closure. Also, we do not know whether the strictness condition is necessary in the theorem. We conclude this section with the promised proof.

Proof of 3.1. We assume that D does so define tc, and derive a contradiction. Our assumption implies that for every ρ as above, and every model of $Comp(D \cup F_\rho)$, the interpretation of tc in the model is the transitive closure of ρ. By strictness, we can have at most one of $tc \geq_{+1} p$ and $tc \geq_{-1} p$; from now on, we shall assume $tc \geq_{+1} p$, since the other possibilities can be refuted by a similar (but easier) argument. We may further assume that $tc \geq q$ for all predicates appearing in D, since, following [14], if we succeed in building a model for all predicates on which tc depends, we may always expand it later to correctly interpret also the predicates on which tc does not depend. Likewise, from now on in this proof, we may ignore all other predicate symbols, and just concentrate on building models which interpret the predicate symbols occurring in D. For each such q, we may define a *sign* $S(q)$, where $S(q)$ is the $i \in \{+1, -1\}$ such that $tc \geq_i q$; so, $S(tc) = S(p) = +1$. We may use S, as in [14], to convert 3-valued models to 2-valued models – namely, if an atom has value $\mathbf{u}$, its value is changed to $\mathbf{t}$ if the sign of its predicate symbol is $+1$, and to $\mathbf{f}$ if the sign is -1.

Since there are no function symbols, the construction of models is a little simpler than in the general case. Let C be the set (possibly empty) of constant symbols used in the program D. Suppose A is a set of constants disjoint from C. A *base structure built from A* is a 3-valued structure, $\mathcal{A}_0$, whose domain of discourse is $A \cup C$, and in which all predicates except (possibly) p are everywhere $\mathbf{u}$. As a special kind of base structure, if ρ is a relation on A, we may interpret p to give $p(a, b)$ value $\mathbf{t}$ if $a \rho b$ and $\mathbf{f}$ otherwise; call this structure $\mathcal{A}_0(\rho)$. For any such A, let T be the 3-valued van Emden - Kowalski operator [11, 13, 14] associated with the program D; as in [14], we define T to affect only the interpretations of predicates other than p (recall that clauses in D do not have p as their head predicate symbol). Starting with some base structure, $\mathcal{A}_0$, we may iterate T to produce a sequence of structures, $\mathcal{A}_0, \mathcal{A}_1, \ldots$ (where $\mathcal{A}_{m+1} = T(\mathcal{A}_m)$); all these structures interpret p the same way, but may differ in their interpretations of other predicates. Since there are no function symbols, if A is finite then there is some finite n such that $\mathcal{A}_n$ is a model of $Comp(D)$. If we start with an $\mathcal{A}_0(\rho)$, then, in addition, $\mathcal{A}_n(\rho)$ will be a model of $Comp(D \cup F_\rho)$.

Now, for each finite K, let A_K be the set of constants $\{a_1, \ldots, a_K\} \cup \{b_1, \ldots, b_K\}$, and we consider three possible base structures built from A_K. Let ρ_K be the relation

$$\{(a_i, a_j) : 1 \leq i, j \leq K\} \cup \{(b_i, b_j) : 1 \leq i, j \leq K\} \quad .$$

Let σ_K be the relation

$$\{(a_i, a_{i+1}) : 1 \leq i < K\} \cup \{(b_i, b_{i+1}) : 1 \leq i < K\} \cup \{(a_K, a_1), (b_K, b_1)\} \quad .$$

Let $\mathcal{B}_0^K$ be the 3-valued structure built from A_K in which all $p(a_i, a_j)$ and $p(b_i, b_j)$ have value $\mathbf{u}$, and p is $\mathbf{f}$ on all other pairs.

The transitive closure of σ_K, σ_K^*, is ρ_K, which is already transitive. The fact, from model theory, that transitive closure is not definable, can be stated with respect to the σ_K as follows:

3.2 Lemma. There is no first-order formula, $\phi(x, y)$, which uniformly (independently of K) defines σ_K^* in the structures $\mathcal{A}_0(\sigma_K)$. ∎

This can be proved either by a compactness argument, or by a direct quantifier elimination, as in Aho-Ullman [1], which proves a slightly different result. We shall eventually use 3.2 to derive a contradiction.

First, starting from $\mathcal{B}_0^K$, the fixedpoint $\mathcal{B}_n^K$ must have the values of $tc(a_i, b_j)$ being $\mathbf{f}$; since if $tc(a_i, b_j)$, were $\mathbf{u}$ or $\mathbf{t}$, we could apply S and obtain a 2-valued model of $Comp(D \cup F_{\rho_K})$ in which $tc(a_i, b_j)$ were $\mathbf{t}$, contradicting the assumption that D correctly defines transitive closure. At first sight, n would appear to depend on K. However, observe that in all the intermediate $\mathcal{B}_m^K$, all relations must be invariant under permutations of the a_i and the b_j; since we are considering only finitely many predicates, the number of such invariant 3-valued structures is bounded independently of K; thus there is actually an n independent of K such that $\mathcal{B}_n^K$ is a fixedpoint of T. Consider this n to be fixed from now on.

By monotonicity of T, it follows that each $tc(a_i, b_j)$ has value $\mathbf{f}$ in $\mathcal{A}_n(\sigma_K)$ as well. Perhaps n is not large enough for $\mathcal{A}_n(\sigma_K)$ to be a fixedpoint, but the $tc(a_i, a_j)$ and $tc(b_i, b_j)$

must be either $\mathbf{t}$ or $\mathbf{u}$ in $\mathcal{A}_n(\sigma_K)$, since these values eventually become $\mathbf{t}$. It follows that the transitive closure, σ_K^*, is first-order definable on $\mathcal{A}_0(\sigma_K)$ (in a way which does not depend on K); namely, it is defined by a formula, $\phi(x, y)$, which "says" that the truth value of $tc(x, y)$ in stage n of iterating T is not $\mathbf{f}$. This contradicts Lemma 3.2. ∎

Observe that Theorem 3.1 would remain true even if we strengthened the logical consequence semantics to consider only Herbrand models of $Comp(P)$, since the proof only deals with how the predicate tc behaves on the constant symbols used in the definition of p. Also, in the literature, there are two distinct definitions of "Herbrand model", depending on whether we build it using all possible Prolog constants and functions, or using only the ones appearing in the program and the query; see Shepherdson [25] for further discussion. It is easy to see that Theorem 3.1 will be true under either interpretation.

§4. **EXTENDING THE LOGICAL CONSEQUENCE SEMANTICS.** There are a number of Prolog constructs which, although usually though of as going beyond pure logic, can in fact be considered to be within the logic. As we point out below, this requires revisions to the definition of $comp(P)$ if even the most elementary completeness results are to hold. Also, this requires, for soundness, that the Prolog evaluation (SLDNF) be revised to return error messages under a greater proportion of circumstances than it now does; this is an extension of the classical requirement that Prolog *flounder* [8] after an improper call to negation. It is important to emphasize that floundering is not the same as failure (returning "no").

We first consider a numeric predicate, the Prolog is. Syntactically, this is simply a binary predicate symbol, and the semantics is usually handled informally as follows: In addition to thc program, P, we assume that there is an implicit set of facts, F_{is}, containing all clauses of the form $is(x, \alpha)$ such that x is a machine representable number and α is a ground numeric term whose value is x. We then consider the semantics to be defined with respect to the set of clauses, $P \cup F_{is}$. Numbers themselves are considered constants. Since is now has an obvious natural interpretation as a relation on the Herbrand universe, the canonical model approaches have a trivial extension to this new situation.

However, there are problems for the logical consequence semantics, both declaratively and procedurally, brought on by the fact that the (implicit) program, $P \cup F_{is}$, now contains infinitely many facts about is (since there are infinitely many terms). Procedurally, we have a problem because there is no natural order in which to backtrack through solutions to a query such as ?$-$ $is(X, Y)$; likewise, we cannot say that ?$-$ $is(X, Y), \neg p(X, Y)$ fails *finitely* even if $p(x, \alpha)$ happens to be true for all ground x and α. So, we must revise our definition of SLDNF to cause these queries to flounder, and we must revise the definition of *allowcd* to not includc thcse queries. In this sense, $is(X, Y)$ behaves very much like a non-ground negative literal. The procedural problems are echoed in the declarative semantics, since the completed definition of is would be an infinitely long sentence; so we must revise the definition of $Comp(P)$ as well.

The appropriate revisions (below) are invoked by declaring the predicate is to be *special*, with the second place declared *unbounded.* More generally, we explain what it means for an n-place predicate, p, to be *special*, with places i through n *unbounded.* First, we mean that in the user's program, P, we do not allow any clauses with p as their head predicate symbol. Second, we consider that P is always augmented by a fixed, possibly infinite, set, F_p, of facts (ground atomic clauses) about p. Third, we modify the definition of $Comp(P \cup F_p)$ by making the completed definition of p contain the following: F_p, plus all $\neg p(\alpha_1, \ldots, \alpha_n)$ such that $p(\alpha_1, \ldots, \alpha_n)$ is ground and is not in F_p, plus, if $i > 1$, all sentences of the form

$$\forall X_1 \ldots X_{i-1}\big(p(X_1, \ldots, X_{i-1}, \alpha_i, \ldots, \alpha_n) \iff$$

$$\bigvee\{(X_1 = \beta_1 \wedge \cdots \wedge X_{i-1} = \beta_{i-1}) : \; p(\beta_1, \ldots, \beta_{i-1}, \alpha_i, \ldots, \alpha_n) \in F_p\}\big)$$

whenever $\alpha_i, \ldots, \alpha_n$ are ground terms; in this case, we demand that for each ground $\alpha_i, \ldots, \alpha_n$, there are only finitely many such $\beta_1, \ldots, \beta_{i-1}$ (if there are none, then the empty disjunction is replaced by *false*). Fourth, in the revised definition of SLDNF, we demand that whenever a literal with p as its predicate is chosen for evaluation, then that literal must be ground in all places with respect to which p is special; as usual, if there is no legal choice of a literal to evaluate, the evaluation flounders.

For example, a typical sentence in the completed definition of *is* would be $\forall X(X \textit{ is } 2+3 \iff X = 5)$. Procedurally, SLDNF must flounder if asked to evaluate something like $5 \textit{ is } Y$. With this in mind, standard Prolog is sound in its handling of arithmetic, if one ignores roundoff and overflow problems. However, it gives up (returns an error) in more places than it needs to; for example, it would be sound to fail with $?-\ X \textit{ is } c$, rather than return an error as most Prolog systems do.

Likewise, numeric comparison operators such as $=:=$ and $<$ may be considered to be special with both places unbounded. This approach to semantics is general enough to allow various implementations of numbers. There could be finitely many integers (say, all with B bits, with $+$ and $*$ defined modulo 2^B), or there could be infinitely many integers (i.e., implementing infinite precision arithmetic). Or, we could consider our numbers to be floating point representations.

A similar discussion can be given for term comparison predicates, such as $\alpha @< \beta$ (which says that α precedes β lexically), except that most Prolog systems are not longer sound here, since they allow comparison of variables. To borrow the standard example from negation, if the program contains the single statement, $p(b, a)$, then most Prologs (e.g., C-Prolog – see the manual, [20]) will succeed with $?-\ X @< Y, p(X, Y)$, and fail with $?-\ p(X, Y), X @< Y$; this cannot be sound under any semantics in which $\wedge$ is commutative. This problem was already pointed out by Naish [19], and is addressed correctly in NU-Prolog [26] (this uses a different symbol, *termCompare*, for the logically correct term comparison predicate, and maintains the old $@<$ with its incorrect semantics). In our notation, we declare $@<$ to be special with both places unbounded, so that queries such as $?-\ X @< Y$ should flounder, not return "yes". Because of the unsound behavior of $@<$, the C-Prolog manual is probably correct in listing $@<$ as a "meta-logical" predicate, along with *var* and *nonvar*; i.e., the procedural behavior of these predicates depends on the binding environment at the time they are called, and they have no declarative meaning.

Note that our notation allows that a special predicate have *all* of its places unbounded, but not that it have *none* of its places unbounded; this latter situation is of no interest here. That is, a special predicate with *no* unbounded places is simply a predicate which the user is not allowed to define, but which behaves as if it were defined by a fixed finite set of clauses. Since the completed definition would be formed in the standard way for these predicates, there is no semantic issue here, and their introduction is simply a matter of *convenience*. Examples of these are *true* (whose implied definition is $\{true.\}$ – that is, it always succeeds), *fail* (whose implied definition is empty – that is, it always fails), and *repeat* (whose implied definition is $\{repeat :- \ repeat. \ \ repeat.\}$). There is some confusion in the literature about the proper status of such predicates. For example, the C-Prolog manual [20] correctly lists *true* under the heading "convenience", but lists *repeat* and *fail* under "extra control", along with some declaratively meaningless control features such as cut and unsound negation.

To state our completeness result, we must make first the appropriate modification in the definition of *allowed*. Let us consider first a few examples. The query $?- \ X \ is \ 3+7$ should certainly be allowed. So should $?- \ q(Y), X \ is \ Y$, since, assuming all the program clauses are allowed, $q(Y)$ should succeed only with ground values for Y (see [7,14]). Likewise, the program clause $earns\_more(X1, X2) :- \ salary(X1, S1), salary(X2, S2), S2 < S1$ should be allowed. The query $?- \ X \ is \ Y, Y \ is \ X$ should not be allowed, but $?- \ X \ is \ 3+7, \neg p(X)$ should be. With these examples in mind, we give a precise definition. Call a variable, X, *focused* in a literal, λ, iff λ is positive, X occurs in λ, and (in the case that the predicate of λ is special), all terms in unbounded places in λ are ground. Call a query clause *allowed* iff every variable occurring in a negated literal or in an unbounded place in a special predicate in the clause is focused in some (other) literal in the clause. Call a program clause *allowed* iff its body is allowed and every variable occurring in the head also occurs in the body.

As another example, the program in the third example in the discussion of Theorem 3.1 is allowed.

In the following theorems, we assume that the appropriate revisions have been made in $Comp(P)$ and SLDNF for any predicates having been declared special. As usual, we state our official result with respect to the 3-valued semantics, but remark that this reduces to the 2-valued semantics under the appropriate strictness conditions, as discussed in §2. That is, if P is call-consistent and is strict with respect to ϕ, and $\forall\phi\sigma$ is a 2-valued consequence of $Comp(P)$, then $\forall\phi\sigma$ is a 3-valued consequence of $Comp(P)$. The proof is as in [14].

4.1 Theorem. SLDNF is sound with respect to the 3-valued semantics; i.e., if SLDNF returns a substitution σ for a query ϕ, then $\forall\phi\sigma$ is a 3-valued consequence of $Comp(P)$, and if SLDNF fails the query ϕ, then $\forall\neg\phi$ is a 3-valued consequence of $Comp(P)$.

4.2 Theorem. If P and ϕ are allowed, then SLDNF is complete; that is, if $\phi\sigma$ is a 3-valued consequence of $Comp(P)$ (which implies that $\phi\sigma$ is ground), then SLDNF will return σ, and if $\forall\neg\phi$ is a 3-valued consequence of $Comp(P)$, then SLDNF finitely fails ϕ.

As with most soundness proofs, Theorem 4.1 is proved by an easy induction. Theorem 4.2 requires some modifications to the proof in [14]; we provide some more details at the end of this section.

Observe that one must use some care in the formulation of the completion in order to get a theorem such as 4.2. It is also important that our semantics allows non-Herbrand models. For example, let P be the program

q :− $p(X), X$ *is* $X + 1$
$p(X)$:− $p(X)$

P is allowed by our definition, and the query ?− $\neg q$ does not produce an answer under our SLDNF, so it is important that $\neg q$ not be supported by the semantics, as it would be if we naively copied the standard definition and took the completed definition of *is* to be a sentence such as

$$\forall X \forall Y (X \textit{ is } Y \iff \bigvee_i (X = x_i \wedge Y = \alpha_i)) \quad ,$$

where the α_i enumerate either all or a finite subset of numeric terms, and each x_i is the numeric value of α_i. Our definition of $Comp(P)$ allows a non-Herbrand model which contains an object, b, for which (b *is* $b + 1$) and $p(b)$ (and hence also q) are true; such a b cannot be in the Herbrand universe. Thus, our formal semantics corresponds with what Prolog does, in that it does no logical inferencing about the consistency of arithmetic equations, but simply uses the built-in machine evaluations as a black box.

Although no one would ever write the program P, we feel that elaborations on it form a class of examples which arise in programming practice. For the trivial $p(X)$:− $p(X)$, one could substitute a more complex recursive definition of p for which ?− $p(X)$ fails to bind X. For the trivial $X + 1$, one could substitute an arbitrarily complicated numeric expression. One might suggest a different semantics, in which numeric variables are intended to range over the *standard* real numbers, and in which the interpreter would realize that $X = X + 1$ has no solution and fail; in fact, the system CLP(R) [12] does something like this (for arbitrary collections of linear equations and inequalities); however, this cannot work in general situations, since the question of the existence of a solution $X = \alpha$ is expensive to decide for polynomial α, and undecidable if α is allowed to contain transcendental functions. To extend our semantics to CLP(R), perhaps the right approach would be expand $Comp(P)$ to contain the (decidable) first-order theory of linear algebra.

The partial completeness result in Theorem 4.2 indicates that the semantics is on the right track, there is much to be done in the following two directions.

First, the completeness results should be strengthened to include a greater proportion of the kinds of programs people write in practice; this problem was already apparent with the results in pure logic [7, 14], but is even more noticeable here. Perhaps the results should take into account the kind of query being posed. Often, one programs chains of computations which can be seen, by ad hoc arguments, to be complete for certain classes of queries. For example, a quadratic equation solver may be programmed by the non-allowed definition,

$solve(A, B, C, S1, S2)$:−
  $\neg(A =:= 0)$,
  D *is* $sqrt(B * B - 4 * A * C)$, $D \geq 0$,
  $S1$ *is* $(-B + sqrt(D))/2/A$, $S2$ *is* $(-B - sqrt(D))/2/A$.

It is easy enough to see here that completeness holds for queries ?− $solve(\alpha, \beta, \gamma, S1, S2)$, where α, β, γ are ground, but it is not so easy to find a simple and elegant sufficient condition which covers all programs of practical interest.

Second, although we have given a logically coherent semantics for these special predicates, it is not the only possible one. In the case of @<, for example, a better procedural semantics would be the one suggested by Naish [19], in which the query ?$-$ $g(X)$@< $f(Y)$ is allowed to fail rather than flounder. There is, in fact, a more general phenomenon at work here, which we now describe, although we do not know if it leads to better general completeness results. If p is an n-place predicate, call a *p-atom* any atomic formula of the form $p(\alpha_1, \dots, \alpha_n)$, where the α_i are terms, not necessarily ground. If S is a set of $p-$ *atoms*, call $S^0 \subseteq S$ a *base* for S iff every atom in S is an instance of some atom in S^0. Declaring p to be *special* means that the user is not allowed to define p, that the system has implicitly defined p by a (possibly infinite) set, F_p, of p-atoms, and furthermore, that the system has partitioned the set of all p-atoms into two subsets, called *floundering* and *non-floundering*. For each p-atom, γ, let U_γ be the set of all $\delta \in F_p$ such that γ and δ are unifiable (after renaming δ with disjoint variables). We require that for each non-floundering p-atom, γ, there is a *finite* base, S_γ, for U_γ. A possible special case would be that $U_\gamma = S_\gamma = \emptyset$, but even if U_γ is finite, or even empty, it is not required that γ be non-floundering; for example, with *is*, $U_{(X \ is \ X+1)} = U_{(2 \ is \ 2+1)} = \emptyset$, but $(X \ is \ X+1)$ is floundering and $(2 \ is \ 2+1)$ is non-floundering. We also require that if γ is non-floundering, then so is $\gamma\sigma$ for every substitution σ. Procedurally, SLDNF is revised so that if it chooses a non-floundering p-atom, γ, as the literal to evaluate, then it examines the (finitely many) unifications with γ and members of S_γ. Declaratively, we now revise the completed definition of p to contain, for each non-floundering p-atom γ, a sentence ϕ_γ defined as follows: Say $X_1, \dots, X_m$ are the variables in γ, $S_\gamma = \{\delta_1, \dots, \delta_s\}$, and σ_i is an mgu of γ and δ_i, chosen so that $\gamma\sigma_i$ uses only variables taken from a set, $\{Y_1, \dots, Y_k\}$, which is disjoint from $\{X_1, \dots, X_m\}$. Then ϕ_γ is

$$\forall X_1 \dots X_m (\gamma \iff \bigvee_{i=1}^{s} \exists Y_1 \dots Y_k (X_1 = X_1\sigma_i \wedge \dots \wedge X_m = X_m\sigma_i)) \quad .$$

As usual, the empty disjunction is replaced by *false*. It is easily seen that the definition of ϕ_γ is independent of the particular choice of the finite base S_γ; that is, a different choice, S'_γ, will lead to a ϕ'_γ which, using the equality axioms, may be proved equivalent to ϕ_γ. There is a certain redundancy in our definition; it would be equivalent to have these ϕ_γ just for γ ranging over a base for the non-floundering p-atoms.

One could even generalize the notion of *special* still further to include the situation where the implicit definition of p need not consist merely of unit clauses, but we do not see an application of this generalization.

In the case of @<, α@< β is non-floundering iff $\alpha\sigma$ compares lexically the same way to $\beta\sigma$ for all ground substitutions σ, in which case α@< β is put into $F_{@<}$ iff these $\alpha\sigma$ are lexically less than the corresponding $\beta\sigma$. Two other predicates common in Prologs which may be cast in this light are $integer(X)$ (testing whether X is an integer) and $name(X, L)$ (X is a constant symbol and L is the list of ASCII codes of the characters with which X is printed). $F_{integer}$ is the obvious set of ground atoms. As with @<, $integer(\alpha)$ would have to be revised to flounder unless α is not a variable; in which case its completed definition is either $\forall\neg integer(\alpha)$ if α is not an integer, or $integer(\alpha)$ if α is an integer. As with @<, Prologs usually fail $integer(X)$, leading to the unsound behavior that $?-\ X\ is\ 3, integer(X)$ succeeds but $?-\ integer(X), X\ is\ 3$ fails. Thus, as discussed above, the C-Prolog manual [20] is correct in listing *integer* and $@<$ as "meta-logical", along with *var* and *nonvar*. Curiously, *name* is also listed as "meta-logical", although its behavior is completely sound. Paraphrasing the manual in our terminology, $name(\alpha, \beta)$ is non-floundering iff either α is a constant symbol or β is a ground list of ASCII codes; F_{name} is the obvious set of ground *name*-atoms. As sample S_γ: $S_{f(X)@<g(X)}$ is $\{f(X)@< g(X)\}$, $S_{g(X)@<f(X)}$ is $\emptyset$, and $S_{name(foo,L)}$ is $\{name(foo, [102, 111, 111])\}$. Note that if $S_\gamma = \emptyset$, then ϕ_γ is equivalent to $\forall\neg\gamma$, and if $S_\gamma = \{\gamma\}$, then ϕ_γ is equivalent to $\forall\gamma$.

Actually, in all our applications, S_γ is either empty or a singleton, but one can imagine situations where it has size larger than 1. For example, one might have an algebraic equation solver, $solve(X, L)$; then $solve(\beta, \alpha)$ is non-floundering iff α is a ground list of floating point numbers, and the procedural semantics will then attempt to unify β with the roots of the polynomial whose coefficients are L; thus, for example, if γ is $solve(X, [1, -3, 2])$, then S_γ has two elements, ϕ_γ is $\forall X(solve(X, [1, -3, 2]) \iff X = 1 \vee X = 2)$, and the query, $solve(X, [1, -3, 2]), X > 3$ fails finitely, since $1 > 3$ and $2 > 3$ fail. In fact, it would be reasonable to allow users to customize such special predicates, programming them in, say, C or Fortran. Many Prologs have some facility for linking in compiled object modules from another language, but we know of none which allows this sort of direct integration with Prolog's backtracking and negation-as-failure mechanism. As with all special predicates, the Prolog semantics would treat *solve* as black box, and would not be responsible for verifying that its implicit definition correctly reflects the intentions of the programmer.

With our extended definitions, soundness (Theorem 4.1) is still true. However, completeness (Theorem 4.2) becomes false, since now the completed definition of a special predicate can now contain a literal of the form $\forall\gamma$ where γ is not ground, so that $\neg\gamma$ becomes false of an infinite set of objects. In the case of $@<$, we have strengthened both the procedural and declarative semantics, but the procedural semantics is not sufficiently strengthened to handle every allowed query supported by the new declarative semantics. For example, if the program is $\{q(X) :-\ q(X)\}$, then the query $?-\ q(X), \neg(f(X)@< g(X))$ now fails by our declarative semantics, but does not fail under our SLDNF.

One could state a version of Theorem 4.2 which handles situations where the implied definition of a special predicate consists only of ground atoms, and in fact we could then weaken the definition of "allowed" somewhat. We do not bother to state such a theorem formally, but an example of this situation would occur with the predicate *name*, which can be used to determine an output in either direction, so it is enough to require that the variables in either one of its arguments be focused. Under the old definitions, *name* would have to be considered to have both its places unbounded, so that $name(X, L), q(X)$ and $name(X, L), q(L)$ would not be allowed query clauses, but in fact, we could consider them to be allowed (assuming q is non-special).

We conclude with a proof of Theorem 4.2. We assume here that the reader has at hand the proof in [14], and just explain what needs to be changed.

Proof of Theorem 4.2. The proof in [14] used the fact, from [13], that the queries supported by $Comp(P)$ in 3-valued logic are exactly those which become true at some finite stage in Fitting's [11] (transfinite) iteration, which was based on the Herbrand universe. This result is false here, as the above example with $(X\ is\ X + 1)$ shows. Rather, we start iterating with a structure, $\mathcal{A}_0$, whose domain of discourse, A, properly extends the Herbrand universe. We assume that $\mathcal{A}_0$ satisfies Clark's equality axioms, that $\mathcal{A}_0$ interprets all non-special predicates to be identically **u**, and that special predicates are interpreted so as to make their completed definitions true. We further assume that the structure contains an element, b, at which all special predicates are undefined as possible. More precisely, say p is special, $\tau_1, \ldots, \tau_n$ are terms, and **v** is the truth value of $p(\tau_1, \ldots, \tau_n)$ when all variables are interpreted as the object b; then we assume **v** is **u** unless all τ_i in unbounded places are ground, in which case **v** is (must be) **f** unless all the τ_i are ground; thus, $(b \ is \ b+1)$ and $(b \ is \ b)$ have value **u**, while $(b \ is \ 2+1)$ has value **f**. Furthermore, we simply assume that the structure $\mathcal{A}_0$ is $\aleph_1$-saturated, so that iterating the appropriate 3-valued T produces a 3-valued model for $Comp(P)$ at stage ω.

Let $\mathcal{A}_{k+1} = T(A_k)$. By induction on k, verify the following: Whenever ϕ is allowed, in variables $X_1, \ldots, X_n$, then there are only finitely many $(a_1, \ldots, a_n)$ in A at which ϕ has truth value **t**, and, furthermore, all these a_i are denotations of ground terms.

The definitions of **R** and **F** must be revised to include the cases where a special literal is evaluated. In the definition of negative rank, one must consider all instantiations of ϕ in $\mathcal{A}$, not just instantiations by ground terms. With these changes in definitions, the inductive hypothesis, $H(\vec{n})$ is stated as before. Regarding the four cases in the proof: Along with R+, we add an R+*, which is the case that we reduce some special literal, $p(\tau_1, \ldots, \tau_m)$, where the τ_i in unbounded places are all ground. Likewise, there is an F+*, where we fail a special positive literal, except here the component of $\vec{n}$ corresponding to this literal is irrelevant. The case R− is unchanged.

The case F− requires more substantial revision. To avoid excess notation, say ϕ is $\alpha, \neg\beta, \gamma$, where α is positive and non-special, β is positive, and γ is positive and special; a similar argument will handle the case where there are several literals of each type. Now, $\vec{n}$ is (n_1, n_2, n_3). As before, we may assume $n_1 = 0$, since otherwise we could reduce by F+, and we may assume $n_3 = 0$, since special predicates do not change in value. We may also assume that γ contains a variable, X, in some unbounded place, since otherwise we could reduce by F+*. As before, it is enough to show that β is ground, but we cannot use the previous argument, since, unlike α, γ is not totally undefined in $\mathcal{A}_0$. Instead, we use our object $b \in A$ described above, and consider the truth values of α, β, and γ at b (i.e., when all variables are interpreted as b). By virtue of X, γ must be **u** at b in $\mathcal{A}_0$, and α must be **u** everywhere in $\mathcal{A}_0$, so (by definition of negative rank), β must be **t** at b in $\mathcal{A}_{n_2}$. If β is special, it must be ground, by our assumptions about b, while if β is non-special, it must be ground, since otherwise, being allowed, it could never become **t** at a non-Herbrand object. ∎

§5. REFERENCES.

[1] Aho, A. V. and Ullman, J. D., Universality of Data Retrieval Languages, *Proceedings of Sixth ACM Symposium on Principles of Programming Languages*, 1979, pp. 110-117.

[2] 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, 1988, pp. 89-148.

[3] Apt, K. R. and Blair, H. A., Arithmetic Classification of Perfect Models of Stratified Programs, Technical Report TR-88-09, University of Texas, Austin, 1988.

[4] Apt, K. R. and Pugin, J.-M., Management of Stratified Databases, Technical Report TR-87-41, University of Texas, Austin, 1987.

[5] Balbin, I., Port, G. S., and Ramamohanarao, K., Magic Set Computation of Stratified Databases, Technical Report 87/3, University of Melbourne, 1987.

[6] Bancilhon, F. and Ramakrishnan, R., Performance Evaluation of Data Intensive Logic Programs, in J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufman, Los Altos, 1988, pp. 439-517.

[7] Cavedon, L., and Lloyd, J. W., A Completeness Theorem for SLDNF Resolution, Technical Report CS-87-06, University of Bristol, 1987, to appear in *J. Logic Programming.*

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

[9] Clocksin and Mellish, *Programming in Prolog*, Springer-Verlag, 1981.

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

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

[12] Jaffar, J. and Lassez, J-L., Constraint Logic Programming, *Proceedings Fourteenth ACM Symposium on Principles of Programming Languages*, Jan. 1987, pp. 111-119.

[13] Kunen, K., Negation in Logic Programming, *J. Logic Programming* 4:289-308 (1987).

[14] Kunen, K., Signed Data Dependencies in Logic Programs, Technical Report #719, University of Wisconsin, 1987, to appear in *J. Logic Programming.*

[15] 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.

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

[17] Lloyd, J. W., *Foundations of Logic Programming*, Second Edition, Springer-Verlag, 1987.

[18] 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).

[19] Naish, L., Negation and Control in Prolog, Lecture Notes in Computer Science #238, Springer-Verlag, 1986.

[20] Pereira, F. and Tweed, C. (eds.), *C-Prolog User's Manual – Versions* 1.5 *and* 1.5^{+}, University of Edinburgh, 1987.

[21] Przymusinski, T. C., On the Declarative Semantics of Deductive Databases and Logic Programs, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufman, Los Altos, 1988, pp. 193-216.

[22] Przymusinski, T. C., Non-monotonic Reasoning vs. Logic Programming: A New Perspective, *to appear.*

[23] Sato, T., On Consistency of First Order Logic Programs, Technical Report TR-87-12, Electrotechnical Laboratory, 1987.

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

[25] Shepherdson, J.C., Negation in Logic Programming, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufman, Los Altos, 1988, pp 19-88.

[26] Thom, J. A., and Zobel, J. (eds.), NU-Prolog reference manual (ver. 1.1), Technical Report 86/10, Machine Intelligence Project, University of Melbourne, 1986.

[27] Van Gelder, A., Negation as Failure Using Tight Derivations for General Logic Programs, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufman, Los Altos, 1988, pp. 149-176.