Negation As Refutation

Melvin Fitting
MLFLC@CUNYVM.CUNY.EDU
Dept. Mathematics and Computer Science
Lehman College (CUNY), Bronx, NY 10468
Depts. Computer Science, Philosophy, Mathematics
Graduate Center (CUNY), 33 West 42nd Street, NYC, NY 10036 *

Abstract

A refutation mechanism is introduced into logic programming, dual to the usual proof mechanism; then negation is treated via refutation. A four-valued logic is appropriate for the semantics: true, false, neither, both. Inconsistent programs are allowed, but inconsistencies remain localized. The four-valued logic is a well-known one, due to Belnap, and is the simplest example of Ginsberg's bilattice notion. An efficient implementation based on semantic tableaux is sketched; it reduces to SLD resolution when negations are not involved. The resulting system can give reasonable answers to queries that involve both negation and free variables. Also it gives the same results as Prolog when there are no negations. Finally, an implementation in Prolog is given.

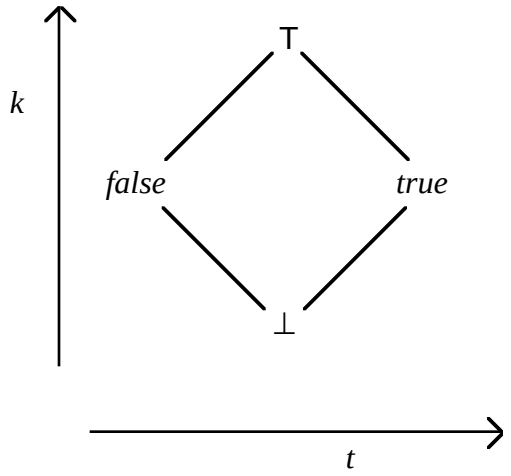
1 Introduction

The most common treatment of negation in logic programming is negation-as-failure. This leads to problems that are now familiar: meanings of programs become difficult to specify; program operators need not reach fixed points in ω steps; queries involving free variables may not behave in expected ways. Consequently alternative notions of negation have been proposed, ranging from simple ideas like delaying evaluation of negative queries containing free variables, to systems built on Intuitionistic Logic. In this paper we propose yet another approach, negation-as-refutation. We believe this avoids the problems just mentioned: there is a simple semantics; the underlying logical structure is of independent interest; and an efficient SLD-like implementation exists. Indeed, we have written such an implementation, under the name Q-Log. (The reason for this name will be explained below.)

Accepting negation-as-refutation forces a rethinking of fundamentals. Classical logic programming is two-valued: statements have as values *true* or *false*. But one can argue that even in a setting without negations, this is too simple. Some queries to some programs may lead to infinite regress,

*Research partly supported by NSF Grant CCR-8702307 and PSC-CUNY Grant 6-67295.

Figure 1: The Logic $\mathcal{FOUR}$



and so a value of $\bot$ (undefined) is reasonable to consider. In fact, this leads to an interesting and coherent semantics which has been developed rather fully in [4], [5], [10] and [11].

Now, a query Q can succeed or fail, but can not do both, so negation-as-failure is inherently consistent. If we introduce a refutation mechanism that stands apart from a provability mechanism, it is possible that some query Q may be both provable and refutable. Consequently we are forced to introduce a fourth truth value, $\top$, or overdefined. Thus our semantics will be based on a four-valued logic, hence the name *Q-Log*, with the Q representing Quad, to suggest four.

2 Belnap's Logic

Belnap introduced exactly the logic we need in [2]; see also [16]. We call it $\mathcal{FOUR}$. More recently, Ginsberg introduced the notion of *bilattice*, with Belnap's logic as the simplest example [7], [8]. In [6] we showed that a satisfactory logic programming semantics could be developed using any bilattice that met certain natural interlacing conditions. $\mathcal{FOUR}$ meets these conditions and, it turns out, will serve us well here.

We present a brief sketch of results about $\mathcal{FOUR}$ that are, in fact, general bilattice theorems. This will serve to lay the semantical foundation for Q-Log.

Following Ginsberg, the truth values of $\mathcal{FOUR}$ are displayed in Figure 1 in a double Hasse diagram. A move from left to right may be thought of as an increase in the degree of truth or a decrease in the degree of falseness. Thus $\bot$ is less false than $false$, and $\top$ is more true than $false$. We write $x \leq_t y$ if y is to the right of (or identical with) x, so $\bot \leq_t false$, for instance. A move upward represents an increase in information or knowledge; we use $\leq_k$ to symbolize this ordering. Both $false$ and $true$ represent more knowledge than $\bot$, which essentially is complete absence of information, so $\bot \leq_k false$ and $\bot \leq_k true$. And so on.

Both orderings, $\leq_t$ and $\leq_k$, give $\mathcal{FOUR}$ the structure of a complete lattice. We will use the notation $\wedge$ and $\vee$ for finite meet and join in the $\leq_t$ ordering, and $\bigwedge$ and $\bigvee$ for arbitrary meet and join. In the $\leq_k$ ordering we will denote finite meet and join by $\otimes$ and $\oplus$, arbitrary meet and join by $\prod$ and $\sum$. The two lattice orderings are closely interconnected. The knowledge operations, $\otimes$ and $\oplus$ are monotone with respect to the truth ordering $\leq_t$, and the truth operations $\wedge$ and $\vee$ are monotone with respect to the knowledge ordering $\leq_k$. This is called the *interlacing* condition in [6]. Further, all possible distributive laws hold. For instance, $x \otimes (y \vee z) = (x \otimes y) \vee (x \otimes z)$. Ginsberg calls this a *distributive* bilattice.

One can also define a natural negation operation. Think of members of $\mathcal{FOUR}$ as *sets* of ordinary truth values: $\bot$ is the empty set, $\top$ is the set consisting of both *true* and *false*, while the members of $\mathcal{FOUR}$ denoted *true* and *false* are singleton sets of the corresponding classical truth values. Now the negation operation on $\mathcal{FOUR}$ is obvious: apply classical negation to every member of the set version. This gives us $\neg\bot = \bot$, $\neg false = true$, $\neg true = false$ and $\neg\top = \top$. It is easy to verify that $\wedge$ and $\vee$ are dual with respect to $\neg$; that is, the usual DeMorgan laws hold. Likewise for $\bigwedge$ and $\bigvee$. On the other hand, each of $\otimes$, $\oplus$, $\prod$ and $\sum$ is self dual. For instance, $\neg(a \otimes b) = \neg a \otimes \neg b$.

The $\leq_t$ operations are natural generalizations of the familiar classical ones. For instance, if $\wedge$, $\vee$ and $\neg$ are restricted to *true* and $false$, we get the usual two value truth table behavior. If they are restricted to *true*, $false$ and $\bot$ the behavior is that of Kleene's strong three valued logic [9]. The $\leq_k$ operations are less familiar, but have natural interpretations. $\otimes$ represents a *consensus* operation; take the most information consistent with the two inputs. For instance, if we get both *true* and $false$ in response to a question, the consensus is $\bot$, no information. And indeed, $true \otimes false = \bot$. Likewise $\oplus$ amounts to accepting whatever one is told. For instance, if we get both *true* and $false$ as answers, and we decide to accept both as equally valid answers, then we have too much information, and in fact, $true \oplus false = \top$.

3 Logic Programming Syntax

We sketch a logic programming language Q-Log which is intended to use $\mathcal{FOUR}$ as its space of truth values. What we describe here is a restricted version of what was presented for general interlaced bilattices in [6]. A *program* will be a finite set of clauses. A *clause* will be an expression of the form $H \leftarrow B$, where H, the head, is an atomic formula and B, the body, is any formula built up from atomic formulas using the connectives $\neg$, $\wedge$, $\vee$, $\otimes$ and $\oplus$. We allow free variables, including those that are free in clause bodies but not in clause heads. We do not allow explicit quantification.

4 Fixpoint Semantics

In classical logic programming semantics, degree of truth is basic, [15], [1]. In effect, $false$ is the default value; an atomic formula takes on *true* as its value only if it is forced to do so. This amounts to using the $\leq_t$ ordering. This can be made more precise and a full treatment can be found in [6]. We wish to shift the emphasis to the information content instead. Informally, an atomic formula will lack a classical truth value (more precisely, it will have the value $\bot$) unless one is forced on it.

As observed in [6], this shift admits a more satisfactory treatment of negation than the classical one, essentially because negation is monotone with respect to $\leq_k$ but not with respect to $\leq_t$.

There are several consequences of this shift in emphasis from truth to knowledge, consequences that combine to make the workings of Q-Log quite smooth. In the classical approach, if one has two clauses with the same head, $A \leftarrow B$ and $A \leftarrow C$, this is taken to be equivalent to a truth-functional disjunction. Informally, the two together act like $A \leftarrow B \vee C$. But here we think of clauses as contributing to the *information* we have, rather than to the degree of truth, and so we take $A \leftarrow B$ and $A \leftarrow C$ together as acting like $A \leftarrow B \oplus C$. Likewise, in the classical approach, free variables in clause bodies are thought of as existentially quantified, which corresponds to using the $\bigvee$ operation in $\mathcal{FOUR}$. We will use the $\leq_k$ analog instead, and interpret free variables in clause bodies using the $\sum$ operation. Incidentally, this has one rather remarkable consequence. We noted above that $\sum$ was its own dual with respect to $\neg$. Since we will be able to implement $\sum$ using unification, we will also be able to implement $\neg \sum$ using unification. In other words, unification will serve well in the presence of free variables, even if negative queries arise.

Now a fixpoint semantics for Q-Log is easily sketched. One associates with a program P an operator Φ_P, mapping interpretations to interpretations. (An interpretation is a function assigning truth values in $\mathcal{FOUR}$ to ground atomic formulas.) For an interpretation v, $\Phi_P(v)$ is determined as follows. Calculate truth values for ground instances of clause bodies using v to supply values at the atomic level; assign the resulting values to the corresponding ground instances of clause heads; that assignment determines a new valuation, which we take to be $\Phi_P(v)$. In evaluating bodies, we think of separate clauses with the same head as being combined using $\sum$, as mentioned above. Interpretations can be given an ordering $\leq_k$, induced by the corresponding ordering in $\mathcal{FOUR}$: $v_1 \leq_k v_2$ provided, for each ground atomic formula A, $v_1(A) \leq_k v_2(A)$. One can show Φ_P is monotone with respect to this ordering. The space of interpretations is a complete lattice under this ordering, and so Φ_P has a smallest fixed point, which provides a semantical meaning for the program P ([14]).

The semantical approach sketched above is developed more fully and more generally in [6]. In the special case we are considering here, with $\mathcal{FOUR}$ as the truth value space, and with programs restricted as described above, an additional very important result holds. For every program P the operator Φ_P must be continuous. This implies that one must be able to fully approximate to the least fixed point of Φ_P in ω steps. This contrasts to the usual situation in logic programming based on $\leq_t$, where programs yielding non-continuous operators are easy to write, and closure ordinals can be very high.

Finally we note that Q-Log has a *paraconsistency* property [3] in the sense that inconsistencies don't make a program useless. If some query A evaluates to $\top$ with respect to a program P (using the least fixed point of Φ_P), other queries may still behave in reasonable ways. Inconsistencies remain isolated, unlike in classical logic.

5 Semantic Tableaux

Having sketched the denotational semantics of Q-Log, we turn to the operational one. We need a suitable proof procedure, and we will use Smullyan's semantic tableaux to help provide the basic

machinery [13]. This is not essential — resolution could also be used. Semantic tableaux provide a mechanism that is easy to explain, though, and one that turns out to be quite efficient in practice.

In one version of Smullyan's system, as presented in [13], one works with *signed* formulas, of the form TX or FX, where X is a formula. Intuitively, TX is read as "X is true", and FX as "X is false." Classically, this use of signs is dispensable; one could replace TX by X, and FX by $\neg X$. But their use has a certain aesthetic appeal, as well as making possible the simplification of some proofs about the system. Tableaux are refutation arguments, like resolution. To prove X, one begins with FX and derives a contradiction. Intuitively this says X can not be false, hence it must always be true. Our immediate problem is to fit this two-valued paradigm to a four valued setting.

The most obvious thing is to introduce four signs, corresponding to the four truth values of $\mathcal{FOUR}$. For our purposes, however, this is quite the wrong thing. We have been thinking of $\bot$ operationally as representing the lack of an answer. It is methodologically wrong to introduce into our proof procedure a symbol which, in effect, would represent the 'answer' of no answer. Similar remarks apply to $\top$ too. In fact, we are thinking of classical two valued logic as being 'behind' $\mathcal{FOUR}$, and this has been reflected in our informal discussions all along: the values of $\mathcal{FOUR}$ are thought of as representing an answer of *true* only, or *false* only, or neither, or both. We want our tableau proof procedure to embody this underlying two valuedness.

There is one other informal point to be made before we get down to details. If we receive an answer of *true* to a query, we do not know, on the basis of this information alone, that the appropriate truth value is just *true*. Later on we might discover that an answer of *false* is also possible, and so the real truth value is $\top$. In short, the best we can expect from an answer to a query is a lower bound on the possibilities: at least *true* (*true* or $\top$), rather than exactly *true*; at least *false* (*false* or $\top$) rather than exactly *false*.

Now, remembering that tableaux are refutation arguments, an appropriate reading of the signs T and F is forced on us. We will intuitively read FX as saying X is either *false* or $\bot$, and we will read TX as saying X is either *true* or $\bot$. Then, if we arrive at a contradiction by starting with FX, it follows that X is either *true* or $\top$, that is, at least *true*. Similarly a contradiction deriving from TX will mean X is at least *false*. If contradictions follow from both TX and FX, it tells us X is $\top$, and if contradictions follow from neither, X is $\bot$.

In the Smullyan two-valued system, proofs are trees, and there are rules for growing them. We consider the rules for conjunction as an example. If $TX \wedge Y$ occurs on a branch, both TX and TY may be added to the end. The intuitive justification for this is: $X \wedge Y$ is true if and only if both X and Y are true. Likewise if $FX \wedge Y$ occurs on a branch, the end of the branch may be split, and FX added to one fork and FY to the other. Again the justification is simple: $X \wedge Y$ is false if and only if either X is false or Y is false. As it turns out, these rules are still correct using the four-valued interpretation given above. It is easy to check that, in $\mathcal{FOUR}$, $X \wedge Y$ is either *true* or $\bot$ if and only if both X and Y are either *true* or $\bot$, hence the T rule; there is a similar justification of the rule for $FX \wedge Y$.

As a matter of fact, every one of Smullyan's (propositional) branch extension rules continues to hold under our four-valued reading. And in addition, we can also develop rules to cover $\oplus$ and $\otimes$ as well. It can be checked that $X \oplus Y$ is *true* or $\bot$ if and only if both X and Y are *true* or $\bot$.

Table 1: The Conjunctive and Disjunctive Cases

α	α_1	α_2
$TX \wedge Y$	TX	TY
$FX \vee Y$	FX	FY
$TX \oplus Y$	TX	TY
$FX \oplus Y$	FX	FY

β	β_1	β_2
$FX \wedge Y$	FX	FY
$TX \vee Y$	TX	TY
$TX \otimes Y$	TX	TY
$FX \otimes Y$	FX	FY

Likewise $X \oplus Y$ is *false* or $\bot$ if and only if both X and Y are *false* or $\bot$. That is, both $TX \oplus Y$ and $FX \oplus Y$ will have non-branching rules. Similarly both $TX \otimes Y$ and $FX \otimes Y$ will have branching rules.

Smullyan introduced a system of *uniform notation* in order to condense several similar-appearing rules into a single one [12], [13]. We extend his classification to cover the $\leq_k$ operations. Two categories of formulas are defined, α signed formulas (which behave conjunctively) and β signed formulas, which behave disjunctively. For each α, two *components*, α_1 and α_2, and for each β, two *components*, β_1 and β_2 are defined. All this is given in Table 1.

Now the tableau branch extension rules for the binary connectives are simply these: a branch containing an α may be extended with α_1 and α_2; a branch containing a β may be split, with one fork containing β_1 and the other β_2. The negation rules are trivial: a branch containing $F\neg X$ may be extended with TX; and a branch containing $T\neg X$ may be extended with FX.

A tableau construction has as its goal the production of contradictions. In Smullyan's system, a branch is *closed* if it contains TX and FX for some formula X. Since this intuitively says X is both true and false, the information on the branch is contradictory, and work on the branch can be discontinued. In our four-valued version, this closure rule must be dropped. Under our reading, TX and FX are jointly possible: X has the value $\bot$. In fact, in a reasonable sense there are no tautologies in $\mathcal{FOUR}$. Even a formula like $p \vee \neg p$ is not always *true*. For instance, if p is $\bot$ then $p \vee \neg p$ is also $\bot$. This behavior is inherited from the Kleene three-valued logic. Consequently, we must introduce extra machinery if we are ever to produce closed tableaux.

From now on we add to the language for Q-Log two propositional constants, *true* and *false*. These can appear in formulas constituting clause bodies in programs. These constants are intended to denote the corresponding truth values of $\mathcal{FOUR}$. And now we have straightforward closure rules: a tableau branch is closed if it contains either T*false* or F*true*. If every branch of a tableau is closed, the tableau itself is called *closed*. Informally, a closed tableau for FX establishes that X is at least *true*; a closed tableau for TX establishes that X is at least *false*. This can, of course, be made into a formal result about the tableau system.

One further observation before we return to logic programming issues. In Smullyan's original system, whenever a propositional rule was applied on a branch the formula to which it was applied could be removed; this did not affect completeness. The same applies here. Further, we have dropped Smullyan's closure rule, and this is the only rule that allows any interaction between formulas. The closure rules we have added in its place only involve single formulas. Only the

α-rule adds multiple formulas to a branch. It follows that, in our present four-valued setting, this rule can be replaced by the non-deterministic rule: if α occurs on a branch, either α_1 or α_2 can be added. Since we can also remove formulas to which rules have been applied, the whole tableau structure simplifies considerably: branches need never contain more than a single formula. Of course the price to be paid is the necessity to backtrack and try a tableau involving α_2 in the event that one involving α_1 fails to close. This kind of backtracking is typical of logic programming, though, and suggests a reasonable place for the introduction of some parallelism.

From now on we will assume all four-valued tableaux are constructed in accordance with all the simplifications introduced above, and so branches consist of single formulas. In implementation terms, this means a tableau can be represented as a list of signed formulas, where each signed formula corresponds to a (degenerate) branch.

Finally, we must assign a role in the tableau construction process to the logic program P whose operational semantics we are supposed to be describing. In fact, we think of P as providing a straightforward set of tableau rewriting rules. Suppose we have a tableau Θ, which are thinking of as a list of signed formulas, as mentioned above. Suppose one of the formulas in Θ is TA, and there is a clause in P of the form $A \leftarrow B$. Then the occurrence of TA in Θ can be replaced by an occurrence of TB. Similarly, if Θ contains FA, this can be replaced by FB. More generally, if Θ contains TA_1 (or FA_1), P has a clause $A_2 \leftarrow B$, and θ is an mgu of A_1 and A_2, then TA_1 (or FA_1) may be replaced by TB (FB) in Θ, and θ applied to all formulas in the result. We call this the *P replacement rule*.

Using the list representation of tableaux, as above, closure is dealt with rather easily. Formulas of the forms $Tfalse$ or $Ftrue$ can be removed from the list. And an empty list represents a closed tableau, and hence a successful argument. Represented this way the similarity to SLD resolution is apparent. If the initial tableau consists of the list $[FX]$, a resulting closed tableau, or empty list, constitutes a *proof* of X. A closed tableau produced by starting with $[TX]$ amounts to a *refutation* of X.

6 Basic Results

Theorem 1 *Let P be a program, and A be a ground atomic formula. A has a truth value of* true *or $\top$ in the least fixed point of Φ_P if and only if A has a proof, allowing the P replacement rule. A has a truth value of* false *or $\top$ in the least fixed point of Φ_P if and only if A has a refutation, allowing the P replacement rule.*

This says our denotational and operational semantics for Q-Log agree well. There is still the problem of familiarity, though. We are using a four-valued logic — how will 'ordinary' Horn clause logic programs behave, since one tends to think of them in classical two-valued terms. A pure Horn clause program corresponds to one in our language that allows only the connective $\wedge$ in clause bodies, and that allows *true* but not *false*. We must allow *true* in order to have a counterpart of $A \leftarrow$, which we represent as $A \leftarrow true$.

Theorem 2 *Let P be a program in which clause bodies may not contain $\oplus$, $\otimes$, $\neg$, $\vee$ or false, so that P has a classical semantics, given by the T_P operator. A ground atomic formula A has the*

value true *in the least fixed point of* T_P *if and only if* A *has the value* true *in the least fixed point of* Φ_P.

On the other hand, when negations are present our treatment will be different than that of, say, Prolog. Most notably, we interpret free variables in clause bodies that are not also in heads using $\sum$ rather than $\bigvee$, and $\sum$ is its own dual under $\neg$. This means that unification is appropriate for queries with free variables whether negations are involved or not. But one must remember that it is the information content of clauses, rather than their truth content, that is fundamental. We illustrate things with a very simple example, one that is familiar in the logic programming literature. Consider the following program, E, for the even numbers:

$$\begin{array}{c} \text{even(0)} \leftarrow \textit{true}. \\ \text{even(s(X))} \leftarrow \neg\text{even(X)}. \end{array}$$

We will ask the query $\neg$even(A), that is, we will attempt to construct a closed tableau starting with $[F\ \neg\text{even(A)}]$. The construction is as follows, ignoring useless paths and backtracking.

$[F\ \neg\text{even(A)}]$	
$[T\ \text{even(A)}]$	using a negation rule
$[T\ \neg\text{even(X)}]$	E-replacement rule, $A = s(X)$
$[F\ \text{even(X)}]$	using a negation rule
$[F\,\textit{true}]$	E-replacement rule, $X = 0$
$[\ ]$	closure rule

A closed tableau results, yielding the answer substitution, $A = s(0)$. The reader may wish to try the slightly more complicated query, even(s(X)).

7 The Guard Connective

We have found it useful to introduce into Q-Log a connective that has no direct classical counterpart, a *guard* connective. In classical logic programming there is a standard way to collapse multiple clauses for the same relation into a single, more complicated, expression. As one step of this conversion one turns $p(a) \leftarrow q$ into $p(X) \leftarrow (X = a) \wedge q$. If this revised clause is used in Prolog, which has negation-as-failure, the $\wedge$ connective does not behave quite like a classical conjunction. To succeed with, say, $p(X)$ one must succeed with both components of the conjunction. But to fail with $p(X)$, because of Prolog's left-right evaluation scheme, either one fails with the unification $X = a$, or one succeeds with it, but fails with q. Thus, whether one is attempting to succeed or to fail, one does not try to succeed or fail with q until first succeeding with $X = a$. The equality behaves less like a component of a conjunction than it does like a guard for q. Observations like these have led us to introduce a four-valued guard connective into Q-Log, and we close with a short discussion of its properties.

We use the notation $p : q$ to symbolize the guard connective; read it p *guards* q. The four-valued truth table is very simple. If p is either $\perp$ or *false*, $p : q$ is $\perp$ (if we cannot pass the guard, we get

Table 2: The Guard Connective Cases

β	β_1	β_2
$FX : Y$	FX	FY
$TX : Y$	FX	TY

no information). If p is either *true* or $\top$ the value of $p : q$ is the value of q. The guard connective has several interesting properties. It is monotone with respect to the $\leq_k$ ordering but not with respect to the $\leq_t$ one. A number of useful four-valued identities hold. For instance, the guard connective is associative, and $p : (q : r) = (p \otimes q) : r$. Distributivity laws hold; if $\circ$ is any of $\wedge$, $\vee$, $\otimes$ or $\oplus$ then $p : (q \circ r) = (p : q) \circ (p : r)$. Of possible left distributive laws, we do have $(p \circ q) : r = (p : r) \circ (q : r)$ for $\circ$ either $\otimes$ or $\oplus$. The other two left distributive laws fail, though we do have $(p \wedge q) : r = (p : r) \otimes (q : r)$ and $(p \vee q) : r = (p : r) \oplus (q : r)$. Finally we also have $p : \neg q = \neg(p : q)$.

Tableau rules for the guard connective are easy. Both the T and the F signed cases act disjunctively, so we merely add two more lines to the β table; the extra lines are given in Table 2.

Finally the guard connective does play its intended role, allowing several clauses of Q-Log for the same relation to be combined into one. As an example, the program for the even numbers given earlier is equivalent to the following (assuming reasonable behavior of equality).

$$\text{even(X)} \leftarrow (\text{X} = 0 : \textit{true}) \oplus (\text{X} = \text{s(Y)} : \neg\text{even(Y)})$$

8 Conclusion

The Q-Log system has been implemented in Prolog after making the usual compromises: unification without an occurs check, and a deterministic order of query evaluation. The implementation is given below. It is extremely simple, and behaves well on queries with free variables in the presence of negation. Send an E-mail request to the author if you would like a copy of the implementation to experiment with.

```
/*  Q-Log,
     a logic programming language based on the
     bilattice FOUR, and implemented using an SLD
     generalization based on semantic tableaux.

     Melvin Fitting
     October 20, 1988
*/

/*   Axioms are written using the syntax:
```

```
     <head> if <body>.  <head> is atomic, and
     <body> is any formula built up from atoms,
     including constants true and false, using
     neg (prefix), and, or, otimes and oplus
     (all infix).  The guard connective, :, is
     also allowed in <body>.  In addition,
     there is a built-in binary predicate, eq.
     eq(X, Y) is turned into Prolog's X = Y.
     neg eq(X, Y) is turned into Prolog's
     X \= Y.  A <body> must always be present.

     Axioms are stored in the form of Prolog
     facts: axiom(---).

     To enter a program, issue the Prolog query:
     program.  This will cause prompts for QLog
     clauses, will read them, and will store them
     as indicated above.  To conclude entering a
     program, enter end.

     To ask a QLog program a query Q, issue the
     Prolog query: query(Q).  The query need not
     be atomic.

     To clear a program, issue the Prolog query:
     reset.
*/


/*   Propositional operators are: and, or, neg,
     oplus and otimes.  And, Or, oplus and otimes
     are left associative.  Also there are signs,
     t and f, not accessible to the user.
*/

?-op(100, fy, neg).
?-op(110, yfx, and).
?-op(110, yfx, or).
?-op(110, yfx, oplus).
?-op(110, yfx, otimes).
?-op(140, fx, [t,f]).
?-op(135, xfx, if).
```

```
?-op(132, yfx, ':').

/*  Define the propositional formula types.
*/

conjunctive(t _ and _).
conjunctive(f _ or _).
conjunctive(t _ oplus _).
conjunctive(f _ oplus _).

disjunctive(t _ or _).
disjunctive(f _ and _).
disjunctive(t _ otimes _).
disjunctive(f _ otimes _).
disjunctive(t _ : _).
disjunctive(f _ : _).

negative(t neg _).
negative(f neg _).

atomicfmla(X) :-
     not conjunctive(X),
     not disjunctive(X),
     not negative(X).

/*  components(F, One, Two) :-
        signed formula F has One and Two
        as its components.
*/

components(t X and Y, t X, t Y).
components(f X or Y, f X, f Y).
components(t X oplus Y, t X, t Y).
components(f X oplus Y, f X, f Y).

components(f X and Y, f X, f Y).
components(t X or Y, t X, t Y).
components(t X otimes Y, t X, t Y).
components(f X otimes Y, f X, f Y).

components(f X : Y, f X, f Y).
components(t X : Y, f X, t Y).
```

```
/*  component(F, One) :-
        signed formula F has One
        as its only component.
*/

component(t neg X, f X).
component(f neg X, t X).

/*  closes(Tableau) :- Tableau can be continued
        to closure, allowing recursive calls via
        the program.
*/

closes([]).

closes([f true|Rest]) :-
     closes(Rest).

closes([t false|Rest]) :-
     closes(Rest).

closes([f eq(X, Y)|Rest]) :-
     X=Y,
     closes(Rest).

closes([t eq(X, Y)|Rest]) :-
     X\=Y,
     closes(Rest).

closes([Negation | Rest]) :-
     negative(Negation),
     component(Negation, Positive),
     closes([Positive | Rest]).

closes([Alpha | Rest]) :-
     conjunctive(Alpha),
     components(Alpha, Alphaone, Alphatwo),
       (closes([Alphaone | Rest]);
        closes([Alphatwo | Rest])).

closes([Beta | Rest]) :-
```

```
     disjunctive(Beta),
     components(Beta, Betaone, Betatwo),
     closes([Betaone, Betatwo | Rest]).

closes([A | Rest]) :-
     atomicfmla(A),
     A =.. [Sign, Head],
     Head \= eq(_,_),
     axiom(if(Head, Body)),
     B =.. [Sign, Body],
     closes([B | Rest]).

/*  And now, the program, execute instructions.
*/

program :-
     write('Enter axioms one at a time, '),
     write('in the form'), nl,
     write('P if Q.'), nl,
     write('"end." to terminate.'), nl,
     read_program.

read_program :-
     write('Enter a clause.'), nl,
     read(X),
     X \== 'end',
     assertz(axiom(X)), !,
     read_program.

read_program :-
     nl, nl, nl, nl, nl,
     write('To ask questions, ')
     write('issue queries of the form'), nl,
     write('query(X).'), nl,
     write('To start over, type'), nl,
     write('reset.'), nl.

query(X) :- closes([f X]),
     write('Q-Log Yes'), nl.
query(X) :- write('Q-Log No'), nl, fail.

reset :-
```

```
     retract(axiom(_)), fail.
reset :-
     write('Done.'), nl.
```

References

[1] K. R. Apt and M. H. van Emden, Contributions to the theory of logic programming, *JACM*, pp 841–862, vol 29 (1982).

[2] N. D. Belnap, Jr. A Useful four-valued logic, in *Modern Uses of Multiple-Valued Logic*, J. Michael Dunn and G. Epstein editors, pp 8–37, D. Reidel (1977).

[3] H. A. Blair, V. S. Subrahmanian, Paraconsistent logic programming, *Proc. of the 7th Conference on Foundations of Software Technology and Theoretical Computer Science, Springer Lecture Notes in Computer Science*, vol 287.

[4] M. C. Fitting, A Kripke/Kleene semantics for logic programs, *Journal of Logic Programming*, pp 295–312 (1985).

[5] M. C. Fitting, Partial models and logic programming, *Theoretical Computer Science*, pp 229–255, vol 48 (1986).

[6] M. C. Fitting, Bilattices and the semantics of logic programming, forthcoming in *Journal of Logic Programming*.

[7] M. L. Ginsberg, Multi-valued logics, *Proc. AAAI-86, fifth national conference on artificial intelligence*, pp 243–247, Morgan Kaufmann Publishers (1986).

[8] M. L. Ginsberg, Multivalued Logics: A Uniform Approach to Inference in Artificial Intelligence, *Computational Intelligence*, vol 4, no. 3.

[9] S. C. Kleene, *Introduction to Methmathematics*, Van Nostrand (1950).

[10] K. Kunen, Negation in logic programming, *J. Logic Programming*, pp 289–308 (1987).

[11] K. Kunen, Signed data dependencies in logic programs, forthcoming in *Journal of Logic Programming*.

[12] R. M. Smullyan, A Unifying principle in quantification theory, *Proc. Nat. Acad. of Sci.*, June 1963.

[13] R. M. Smullyan, *First Order Logic*, Springer-Verlag, 1968.

[14] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, *Pacific Journal of Mathematics*, vol 5, pp 285–309 (1955).

[15] M. van Emden and R. A. Kowalski, The Semantics of predicate logic as a programming language, *JACM*, pp 733–742, vol 23 (1976).

[16] A. Visser, Four valued semantics and the liar, *Journal of Philosophical Logic*, pp 181–212, vol 13 (1984).