ABSTRACT

A logic using three truth values (true, false, undefined) is described together with its Horn clause subset and a procedural interpretation. The resulting logic programming language allows clauses to affirm both positive and negative information and can test whether a goal is definitely false or is just not provably true (standard negation by failure) as well as other possibilities including whether it is unknown (cannot be proven either true or false). The major theoretical results characterizing classical logic programs can be carried over to this context, including the equivalence of a programs answer set with the minimal Herbrand universe and least fixed point semantics as well as the correctness and completeness of SLD–resolution. The logic can be easily implemented within existing Prolog interpreters.

I INTRODUCTION

It is well known that expressing negative information in Prolog programs causes difficulties. LLoyd [1984] discusses these in some detail. The basic problem is that the negation of a condition cannot be inferred directly from the clauses of a Logic Program. They provide only information about what is true, not about what is false. This can be circumvented to a certain extent by using the negation as failure rule where a negation is inferred when no positive solution to a goal can be found. The logical justification of negation as failure (NAF) has caused some difficulties, see Flanagan[1986] for the current state of this subject. This paper does not directly address the issue of the logical status of NAF but provides a setting where the programmer can say what she really means by the negation of a concept. The multivalued logic can effectively use NAF for showing that a goal is unknown but its logical basis is no better than that of NAF in classical logic programming.

This paper explores a technique for affirming negative information using a three valued logic (true, false, undefined) rather than the classical true and false. The informal

interpretation given to these values is that true means *can definitely be shown to be true*, false means *can definitely be shown to be false*, and undefined means *cannot be shown to be either true or false*.

An example will show the type of programs which result from this approach. Consider a program used by a tax department to check tax returns. This very likely includes logic to check whether two tax payers are married and if so what their resulting tax category is. A normal data-base expression of who is married might look as follows:

```
married(john,dorothy).
married(brian,sheila).
```

However, in many cases it can be deduced that a particular couple cannot possibly be married, in the proposed system this can be expressed as follows:

```
~married(X,Y):- male(X), male(Y).
~married(X,Y):- female(X), female(Y).
~married(X,Y):- married(X,Z), Z≠Y.
~married(X,Y):- age(X,N), N < 16.
~married(james,laura).
```

(there are many other clauses one might add here).

This still leaves some couples whose status is unknown. In these cases the truth of married(X,Y) is assigned unknown just as in classical logic programming (using NAF) anything not known to be true is assigned false. The negative part of the married predicate can be considered to be an integrity constraint on the positive part of married (or *vice versa*!). It is more useful than the usual integrity constraints because it can be used for computing results.

This data can be used in a number of ways, for example some suspicious tax departments assume that a couple who could possibly be married and who live at the same address are effectively married. This can be expressed in the following rule where +married(X,Y) means could possibly be married (that is, it cannot be proven that X and Y are not married):

```
cohabit(X,Y):- +married(X,Y), same_address(X,Y).
```

It is also possible to conclude that a couple definitely do not cohabit if they are provably not married:

~cohabit(X,Y):- ~married(X,Y).

In some cases the tax department will want to make enquiries about a couples status if they do not know their marital status and they are living at the same address. This is done using ?married(X,Y) which means that married(X,Y) is unknown (that is, it cannot be shown that X and Y are either married or not married):

check_marital_status(X,Y):- ?married(X,Y), same_address(X,Y).

Conversely, if sufficient information is available then the tax return can be processedwithout asking any further questions. Thus if a couple live at the same address and their marital status is known then the return can be processed. The following rule expressing this uses the goal !married(X,Y) which means married(X,Y) is known to be either true of false.

process_tax_return(X,Y):- same_address(X,Y), !married(X,Y).

In a rare moment of generosity the department may also assume that the returns for a couple can be processed if they are not known to be living at the same address. The goal ¬same_address used in the rule means that same_address cannot be proven true (it may be false or unknown).

process_tax_return(X,Y):- ¬same_address(X,Y).

The next section provides a formal description of the logic used including the various logical operators which can be derived and some useful tautologies between them. It also shows how statements in the multivalued logic can be translated into (more complex) statements in classical logic. Section III describes the Horn clause subset of the logic to be

used for programming. It establishes the equivalence of the answer set, the least Herbrand model and the least fixed point. It also notes the importance of a program being satisfiable. Section IV shows the correctness and completeness of an extended form of resolution. Section V extends the execution rules to include the use of negation by failure. Section VI considers how to show that a program is satisfiable. Section VII gives a detailed comparison between the logical operators used here and those used in other three valued logics and concludes by briefly exploring similar extensions to other multivalued logics.

II THE LOGIC

Define a first order language as in [Lloyd, 1984, Chapter 2] with the usual notion of function, predicate, constant and domain. The quantifiers for the language are $\forall$ and $\exists$, the unary connectives are ~ and $\neg$ (two different forms of negation) and the binary connectives are $\wedge$, $\vee$, and $\leftarrow$. (A few more connectives will be introduced later). In constructing an *interpretation* for this language the only difference from classical first order logic is that each predicate is mapped into {true, false, undefined} abbreviated hereafter as {**t**, **f**, **u**} respectively. The truth value of a formula in a particular interpretation can be calculated using the following rules:

(a) $\forall$x**A** has the truth value:

- **t** if for all values of x **A** has the truth value **t**;
- **u** if for all values of x **A** is **t** or **u** and for at least one value of x **A** has the truth value **u**;
- **f** if there is at least one value of x for for which **A** has the truth value **f**.

(b) $\exists$x**A** has the truth value:

- **t** if there is at least one value of x for for which **A** has the truth value **t**.

u if for all values of x **A** has the truth value **f** or **u** and for at least one value of x **A** has the truth value **u**;

f if for all values of x **A** has the truth value **f**.

(c) **~A** , **¬A** , **A∧B**, **A∨B**, **A←B** have the values determined by the following tables:

A	t	u	f
~A	f	u	t

A	t	u	f
¬A	f	t	t

A∧B	A: t	A: u	A: f
B: t	t	u	f
B: u	u	u	f
B: f	f	f	f

A∨B	A: t	A: u	A: f
B: t	t	t	t
B: u	t	u	u
B: f	t	u	f

A← B	A: t	A: u	A: f
B: t	t	u	f
B: u	t	t	t
B: f	t	t	t

Given these computation rules for an interpretation the usual definitions can be made.

Definition

A closed formula is: *valid* if it has the truth value **t** in every interpretation (that is every interpretation is a *model*); *satisfiable* if it has an interpretation which is a model; *unsatisfiable* if it has no model.

Definition

Given two closed formulae **A** and **B** then **A** is a *logical consequence* of **B** if every model of **B** is also a model of **A**.

A direct consequence of these definitions and the truth tables is that **A** is a logical consequence of **B**

iff **A← B** is valid;

iff **B∧¬A** is unsatisfiable.

Implication and Equivalence

The operators introduced above are sufficient to construct a logic programming language. However, it is useful to introduce a few more operators to help with developing the theory of the logic. These are **A↔B**, **A⇔B**, and **A⇐B** with the following truth tables:

A⇔B	A: t	u	f
t	t	u	f
u	u	t	u
f	f	u	t

A⇐B	A: t	u	f
B: t	t	u	f
B: u	t	u	u
B: f	t	t	t

A↔B	A: t	u	f
B: t	t	u	f
B: u	u	t	t
B: f	f	t	t

↔ can be interpreted as a weak form of equivalence, it will be particularly useful when dealing later with statements in clausal form. It can be defined as (**A←B**) ∧ (**B←A**). It is referred to as a *weak* equivalence because the validity of (the closure of) **A↔B** is not sufficient to ensure that **A** and **B** can be substituted for one another in all cases.

⇔ is a strong form of equivalence and if **A⇔B** is valid then **A** and **B** can be interchanged in any expression without altering the value of the expression. **A⇔B** is not equivalent to (**A⇐B**) ∧ (**B⇐A**) although it is equivalent to (**A∧B**) ∨ (**~A∧~B**) ∨ (**?A∧?B**) (? is defined a little later). This is analogous to the classical equivalence between **A⇔B** and (**A∧B**) ∨ (**~A∧~B**).

⇐ can also be interpreted as a strong form of implication, although as noted above it cannot be used to generate ⇔. It is equivalent to (**A∨~B**).

The various quantifiers and connectives are not independent of one another. Some of the

useful equivalences are given below.

$$\sim\forall X\,\mathbf{A} \Leftrightarrow \exists X \sim\mathbf{A} \qquad \forall X \sim\mathbf{A} \Leftrightarrow \sim\exists X\,\mathbf{A}$$

$$\neg\forall X\,\mathbf{A} \Leftrightarrow \exists X \neg\mathbf{A} \qquad \forall X \neg\mathbf{A} \Leftrightarrow \neg\exists X\,\mathbf{A}$$

$$\sim(\mathbf{A} \wedge \mathbf{B}) \Leftrightarrow (\sim\mathbf{A} \vee \sim\mathbf{B}) \qquad \sim(\mathbf{A} \vee \mathbf{B}) \Leftrightarrow (\sim\mathbf{A} \wedge \sim\mathbf{B})$$

$$\neg(\mathbf{A} \wedge \mathbf{B}) \Leftrightarrow (\neg\mathbf{A} \vee \neg\mathbf{B}) \qquad \neg(\mathbf{A} \vee \mathbf{B}) \Leftrightarrow (\neg\mathbf{A} \wedge \neg\mathbf{B})$$

$$\mathbf{A} \leftarrow \mathbf{B} \Leftrightarrow \mathbf{A} \vee \neg\mathbf{B}$$

One consequence of this is that the entire logic could be defined in terms of say, $\forall$, $\sim$, $\vee$ and $\neg$, the other quantifiers and connectives being derived from them.

Starting from this base twelve different unary operators can be derived. These are generated by taking all possible truth tables where **u** is mapped to **t**, **f** or **u** and **t** and **f** are mapped to **t** or **f**. Figure 1 lists them together with some of the logical equivalences between them. The Figure includes the three unary operators, !, ?, and +. These can be defined in terms of the others:

$$!\mathbf{A} \Leftrightarrow \mathbf{A} \vee \sim\mathbf{A}$$

$$?\mathbf{A} \Leftrightarrow \neg\mathbf{A} \wedge \neg\sim\mathbf{A}$$

$$+\mathbf{A} \Leftrightarrow \neg\sim\mathbf{A}$$

It will be seen later that these three as well as the original operators $\sim$ and $\neg$ have useful procedural interpretations and so are singled out with unique names.

These operators can be interpreted informally as follows:

$\sim\mathbf{A}$	**A** is falsifiable
$\neg\mathbf{A}$	**A** is not provable (usual negation by failure)
$?\mathbf{A}$	**A** is unknown (is neither provable nor falsifiable)
$!\mathbf{A}$	**A** is known (it is either provable or falsifiable)
$+\mathbf{A}$	**A** is not falsifiable

$\mathbf{A} \leftarrow \mathbf{B}$ if **B** is provable then **A** is provable.

$\wedge$, $\vee$, $\forall$, and $\exists$ can be interpreted in the usual way as and, or, for all and there exists.

Translation to Classical Logic

It is possible to interpret any sentence in the multi-valued logic as a sentence in ordinary classical logic. This can be done by recognizing that the truth value of a formula can be interpreted as the truth or falsity of two statements in classical logic. Thus **t** can be interpreted as the pair <true, false> (it is provable but not falsifiable), **u** can be interpreted as <false, false> (it is neither provable nor falsifiable) and **f** as <false,true> (it is not provable but is falsifiable). To realize this translation each predicate $p(t_1,...,t_n)$ can be renamed to one of two predicates $p_+(t_1,...,t_n)$ or $p_-(t_1,...,t_n)$ in the classical logic. These two correspond respectively to the idea that p is provable or that p is falsifiable.

The translation can be carried out by three mappings $\mathfrak{C}$, $\mathfrak{P}$ and $\mathfrak{F}$ from the multivalued logic to the classical logic. Informally $\mathfrak{C}$ ensures that no atom is assigned to both true and false simultaneously and $\mathfrak{P}(\mathbf{A})$ means **A** is provable and $\mathfrak{F}(\mathbf{A})$ means **A** is falsifiable. The mappings $\mathfrak{P}$ and $\mathfrak{F}$ are defined recursively by:

$\mathfrak{P}(p(t_1,...,t_n)) = p_+(t_1,...,t_n)$

$\mathfrak{P}(\sim\mathbf{A}) = \mathfrak{F}(\mathbf{A})$

$\mathfrak{P}(\neg\mathbf{A}) = \sim\mathfrak{P}(\mathbf{A})$

$\mathfrak{P}(\mathbf{A}\wedge\mathbf{B}) = \mathfrak{P}(\mathbf{A}) \wedge \mathfrak{P}(\mathbf{B})$

$\mathfrak{P}(\forall x\mathbf{A}) = \forall x\, \mathfrak{P}(\mathbf{A})$

$\mathfrak{F}(p(t_1,...,t_n)) = p_-(t_1,...,t_n)$

$\mathfrak{F}(\sim\mathbf{A}) = \mathfrak{P}(\mathbf{A})$

$\mathfrak{F}(\neg\mathbf{A}) = \sim\mathfrak{F}(\mathbf{A})$

$\mathfrak{F}(\mathbf{A} \wedge \mathbf{B}) = \mathfrak{F}(\mathbf{A}) \vee \mathfrak{F}(\mathbf{B})$

$\mathfrak{F}(\forall x\ \mathbf{A}) = \exists x\ \mathfrak{F}(\mathbf{A})$

$\mathfrak{C}(\mathbf{A})$ is the formula obtained by the conjunction of the statements:

$$\forall x_1,...,x_n \sim(p_+(x_1,...,x_n) \wedge p_-(x_1,...,x_n))$$

for each n-ary predicate symbol p which appears in **A**.

Theorem:

A is unsatisfiable (in the multivalued logic) iff ($\mathfrak{C}(\mathbf{A})$ and $\mathfrak{P}(\mathbf{A})$) is unsatisfiable (in the classical logic).

Proof:

Every multivalued model of **A** has a corresponding interpretation where both $\mathfrak{C}(\mathbf{A})$ and $\mathfrak{P}(\mathbf{A})$ will evaluate to true. Conversely any model of $\mathfrak{C}(\mathbf{A})$ can be mapped to an interpretation of **A** and if the interpretation is a model of $\mathfrak{P}$(A) then it will be a model of **A**.

◊

This result will be useful both in justifying the procedural implementation of the logic described in section IV and in establishing the fixed-point results of the next section.

III HORN CLAUSE SUBSET

A "Horn clause" subset for the logic can be built up as follows (following [Lloyd, 1984]).

Definition

A *literal* is any term of the form **A**, ~**A**, ¬**A**, ?**A**, !**A**, +**A**, ~!**A**, ¬+**A**, ¬¬**A**, or ¬?**A** where **A** is an atom.

This definition allows all the ten possible unary operators in Figure 1 (excluding the trivial true and false). After arriving at the final Horn clause form it will be seen that the compound literals (e.g. ¬+**A**) can be eliminated.

Definition

A *clause* is a formula of the form $\forall(\mathbf{L}_1 \vee ... \vee \mathbf{L}_m)$ where each $\mathbf{L}_i$ is a literal.

It is convenient to rewrite clauses in the more familiar form:

$$\mathbf{L} \leftarrow \mathbf{L}_1 \wedge ... \wedge \mathbf{L}_m$$

which using the definition of $\leftarrow$ is equivalent to the clause:

$$\forall(\mathbf{L} \vee \neg\mathbf{L}_1 \vee ... \vee \neg\mathbf{L}_m)$$

L is referred to as the *head* of the clause and $\mathbf{L}_1 \wedge ... \wedge \mathbf{L}_m$ as the *body* of the clause.

This rewriting requires more justification than in the classical case. It is not clear for example that any clause can be so rewritten. The following results justify the procedure.

Definition

Given some formula **C** which contains an occurrence of the formula **A**, then **A** is said to occur positively in **C** if the translation $\mathfrak{P}(\mathbf{C})$ contains only formulae of the form $\mathfrak{P}(\mathbf{A})$ (and not of the form $\mathfrak{F}(\mathbf{A})$).

Informally a positive occurrence is one which is not nested inside an odd number of occurrences of the ~ operator (or of any occurrences of other operators such as ?, ! or +). For example **A** occurs positively in the following formulae: $\mathbf{A} \vee \mathbf{B}$, $\sim(\sim\mathbf{A} \vee \mathbf{B})$; and does not occur positively in the following formulae: $\sim(\mathbf{A} \vee \mathbf{B})$, $?\mathbf{A}$.

Proposition

If (the closure of) $\mathbf{A} \leftrightarrow \mathbf{B}$ is valid then **B** can replace any positive occurrence of **A** in **C** without affecting the unsatisfiability of **C**.

Proof:

Consider the corresponding replacement in the classical translation of **C**. It is only necessary to consider any interpretation which is a model of $\mathfrak{C}$(**C**). Then $\mathfrak{P}$(**A**) will evaluate to true iff **A** evaluates to **t**. But if **A**↔**B** is valid then **A** evaluates to **t** iff **B** evaluates to **t** (by inspection of the truth table). So $\mathfrak{P}$(**A**) iff $\mathfrak{P}$(**B**). But the unsatisfiability of $\mathfrak{P}$(**C**) is affected only by $\mathfrak{P}$(**A**) so the truth value of $\mathfrak{P}$(**C**) will be unaffected by the replacement. So the unsatisfiability of ($\mathfrak{C}$(**C**) and $\mathfrak{P}$(**C**)) and hence of **C** is unaffected.

◊

Corollary

If **A**↔**B** is valid then any occurrence of **A** as a literal in a clause can be replaced by **B** without affecting the validity of the clause.

Proof:

Each literal in a clause is a positive occurrence.◊

Figure 2 shows how any clause can be translated to an equivalent clause using the appropriate weak equivalences. Note that the ten possible unary operators can be translated to just those 6 operators with simple names (fulfilling the earlier promise that compound unary operators can be dispensed with).

I will now show the equivalence of the least Herbrand model, the least fixed point and the answer set for multi-valued programs. Many of the theorems are essentially identical to those in [LLoyd, 1984] and their proofs are omitted. I will note these by putting the reference number from Lloyd in brackets.

Theorem

If **S** is a finite conjunction of clauses then there is a finite conjunction of (classical) clauses which is unsatisfiable iff ($\mathfrak{C}$(**S**) and $\mathfrak{P}$(**S**)) is unsatisfiable.

Proof:

It is nearly true that ($\mathfrak{C}$(S) and $\mathfrak{P}$(S)) is itself a conjunction of clauses. By inspection $\mathfrak{C}$(S) is a conjunction of clauses and $\mathfrak{P}$(S) will be a conjunction of clauses except when one of the literals is of the form $!p(t_1, ..., t_n)$. Given a clause $\mathbf{L} \leftarrow \mathbf{L}_1 \wedge ... \wedge \mathbf{L}_m$ it will be translated to $\mathfrak{P}(\mathbf{L}) \leftarrow \mathfrak{P}(\mathbf{L}_1) \wedge ... \wedge \mathfrak{P}(\mathbf{L}_m)$. The translation of the six possible literals which can occur in a clause is given by the following table (p is an abbreviation for some n–ary predicate):

$\mathfrak{P}(p) = \quad p_+$

$\mathfrak{P}(\sim p) = \quad p_-$

$\mathfrak{P}(\neg p) = \quad \sim p_+$

$\mathfrak{P}(+p) = \quad \sim p_-$

$\mathfrak{P}(!p) = \quad p_+ \vee p_-$

$\mathfrak{P}(?p) = \quad \sim p_+ \wedge \sim p_+$

The required clauses can be obtained whenever !p occurs by using the (classical) rule that $\mathbf{L} \leftarrow \mathbf{A} \vee \mathbf{B}$ can be split into two equivalent clauses $\mathbf{L} \leftarrow \mathbf{A}$ and $\mathbf{L} \leftarrow \mathbf{B}$.

◊

Although not germain to the development here similar reasoning to the theorem above shows that any multi-valued formula can be translated to a clausal form. This is easily seen by translating the original formula to its classical equivalent, reducing that to clausal form and then translating back to the multi-valued logic. Needless to say this can be done more directly.

Definition

The *Herbrand base* for a (multivalued) language can be obtained in the same way as in a classical logic by constructing all possible ground terms from predicates, functions and constants occurring in the language. A *Herbrand interpretation(model)* can also be identified with an extended subset of the Herbrand base. Let the *extended Herbrand base* be all terms of the form **A** or ~**A** where **A** is some ground atom taken from the Herbrand

base. A Herbrand interpretation I can be identified with a subset of this extended Herbrand base by mapping the term **A** to **t** if **A**∈I , to **f** if ~**A**∈I and to **u** otherwise. For this to be well–defined **A** and ~**A** cannot both occur in I.

Theorem (3.2)

If **S** is a finite set of clauses then if it has a model it has a Herbrand model.

Proof:

Because of the result above that **S** has a classical clausal translation, a Herbrand model of the translation can be constructed and used to generate a (classical) Herbrand model and from this a multi-valued Herbrand model.

◊

Theorem (3.3)

A finite set of clauses is unsatisfiable iff it has no Herbrand models.

◊

Definition

A *program clause* is a clause of the form:

$$\mathbf{L} \leftarrow \mathbf{L}_1 \wedge \ldots \wedge \mathbf{L}_m$$

where **L** is of the form ~**A** or **A** for some atom **A**

and each of the $\mathbf{L}_i$ is of the form **B**, ~**B**, !**B** for some atom **B**.

It will be seen that a program clause can be executed without invoking any negation by failure rule. The operators ¬, +, and ? allowed in a general program clause in general require a negation by failure rule.

Definition

A *goal* is a clause of the form:

$$\leftarrow \mathbf{L}_1 \wedge \ldots \wedge \mathbf{L}_m$$

where each of the $\mathbf{L}_i$ is of the form **B**, ~**B**, !**B** for some atom **B**.

Definition

A *program* is a finite conjunction of program clauses.

Theorem (6.1)

If **P** is a program and $\{M_i\}_{i\in I}$ a non-empty set of Herbrand models for **P** then $\cap_{i\in I}M_i$ is a Herbrand model for **P**.

◊

At this point the theory for multivalued programs differs from classical programs because it is possible that a program has no model (is unsatisfiable). For example the program

~p←
p←

has no model. The standard results apply only when the program has some model.

Theorem

If a program **P** is satisfiable then it has a least Herbrand model denoted M**p**.

Proof:

The set of Herbrand models is not empty so the intersection of all of them is the least Herbrand model.

◊

Theorem(6.2)

If a program **P** is satisfiable then M**p** equals the set of all members of the extended Herbrand base which are logical consequences of **P**.

◊

A mapping T**p** on Herbrand interpretations can be defined as in the classical case.

Definition

Let **P** be a program and I a Herbrand interpretation. Then **A** is a member of T**p**(I) iff **A**←$\mathbf{A}_1$, ..., $\mathbf{A}_n$ is a ground instance of a clause in **P** and $\{\mathbf{A}_1, ..., \mathbf{A}_n\}\subseteq$ I.

Theorem

If a program **P** is satisfiable and I is a Herbrand interpretation then $T_{\mathbf{P}}(I)$ is a Herbrand interpretation.

◊

Theorem (6.5)

If a program **P** is satisfiable then $M_{\mathbf{P}}$ equals the least fixed point of $T_{\mathbf{P}}$ which equals $T_{\mathbf{P}}\uparrow\omega$

◊

While $T_{\mathbf{P}}$ is defined for any program it gives nonsensical results unless **P** is satisfiable. This places the onus on the programmer to ensure that her program is satisfiable. The practicalities of this are discussed in section VI. It is also interesting to note that $T_{\mathbf{P}}$ need not have a greatest fixed point. The entire extended Herbrand base is not a model (unlike the classical case) so a greatest fixed point cannot be constructed by applying $T_{\mathbf{P}}$. For example the program:

p← p

~p← ~p

has the empty set as its least Herbrand model, other models are {p} and {~p} but no model can contain both p and ~p.

IV PROCEDURAL ASPECTS

In order to give a computational reading to a program it is necessary to describe the multivalued analog of resolution.

Theorem

Given a satisfiable program **P** containing a clause $\mathbf{L} \leftarrow \mathbf{L}_1 \wedge ... \wedge \mathbf{L}_l$ then

$\mathbf{P} \wedge \leftarrow \mathbf{M}_1 \wedge ... \wedge \mathbf{M}_m$ is unsatisfiable if

$\mathbf{P} \wedge \leftarrow (\mathbf{L}_1 \wedge ... \wedge \mathbf{L}_l \wedge \mathbf{M}_2 \wedge ... \wedge \mathbf{M}_m)\theta$ is unsatisfiable

provided $\mathbf{M}_1\theta \leftarrow \mathbf{L}\theta$ is valid and θ is a unifier for $\mathbf{M}_1$ and **L**.

Proof:

It is only necessary to consider the propositional case which can be verified by exhaustion of the truth tables.◊

The situation here is a little more complex than in the classical case. There are many more resolutions which are possible because both **A** and ~**A** can occur in a head as well as 3 different operators in the body. The relevant tautologies are:

A←**A**

~**A**←~**A**

A←!**A**

~**A**←!**A**

Figure 3 illustrates the resolution steps permitted by these tautologies as well as some others permitted for other literals than those allowed in a program clause. Note that !**A** represents "is known" that is "is provable or falsifiable" and can be resolved against either **A** or ~**A**. These resolution steps are mirrored by comparable resolutions which can be done in the translation of the program. Thus p will translate to p_+ in a body and can resolve against p_+ in a head, similarly, for ~p and p_-. !p translates to p∨~p and so can resolve against either p_+ or p_-. This observation can be formalized in the following theorem.

Theorem

Every goal **G** has an (SLD-)refutation using a program **P** iff $\mathfrak{P}(\mathbf{G})$ has a classical (SLD–)refutation using $\mathfrak{P}(\mathbf{P})$.◊

Finally the correctness and completeness results for classical SLD-refutations can be carried over to the multi-valued case by the following result.

Theorem

Given a satisfiable program **P** and a goal **G**

$\mathbf{P} \wedge \mathbf{G}$ is unsatisfiable iff

$\mathfrak{P}(\mathbf{P}) \wedge \mathfrak{P}(\mathbf{G})$ is unsatisfiable

Proof:

1) Assume $\mathfrak{P}(\mathbf{P}) \wedge \mathfrak{P}(\mathbf{G})$ is unsatisfiable then $\mathfrak{C}(\mathbf{P} \wedge \mathbf{G}) \wedge \mathfrak{P}(\mathbf{P}) \wedge \mathfrak{P}(\mathbf{G})$ is unsatisfiable so $\mathbf{P} \wedge \mathbf{G}$ is unsatisfiable.

2) Assume that $\mathfrak{P}(\mathbf{P}) \wedge \mathfrak{P}(\mathbf{G})$ has a model, then it must have a Herbrand model (which is also a model of $\mathfrak{P}(\mathbf{P})$). But any subset of a model of $\mathfrak{P}(\mathbf{G})$ is also a model of $\mathfrak{P}(\mathbf{G})$ so $M_{\mathfrak{P}(\mathbf{P})}$ is a model of $\mathfrak{P}(\mathbf{P}) \wedge \mathfrak{P}(\mathbf{G})$. By the premise $\mathfrak{C}(\mathbf{P}) \wedge \mathfrak{P}(\mathbf{P})$ has a model and so a Herbrand model. But any subset of a model of $\mathfrak{C}(\mathbf{P})$ is still a model of $\mathfrak{C}(\mathbf{P})$ so $M_{\mathfrak{P}(\mathbf{P})}$ is a model of $\mathfrak{C}(\mathbf{P}) \wedge \mathfrak{P}(\mathbf{P})$. But, apart from the trivial case when **G** contains predicates not occurring in **P**, $\mathfrak{C}(\mathbf{P}) = \mathfrak{C}(\mathbf{P} \wedge \mathbf{G})$ so $M_{\mathfrak{P}(\mathbf{P})}$ is a model of $\mathfrak{C}(\mathbf{P} \wedge \mathbf{G}) \wedge \mathfrak{P}(\mathbf{P}) \wedge \mathfrak{P}(\mathbf{G})$ and so $\mathbf{P} \wedge \mathbf{G}$ has a model.

◊

V NEGATION BY FAILURE

The upshot of the last section is that a normal Prolog execution strategy can be followed provided the program is consistent and so long as the program clauses contain only goals of the form **A**, **~A** and **!A**. However, some other strategies can be followed because it is known that both **A** and **~A** cannot be simultaneously be true. For example, before executing a goal, say $p(t_1, ..., t_n)$, it may be worthwhile to execute $\sim p(t_1, ..., t_n)$ and fail the original goal if this succeeds. If ~p fails however it is still necessary to execute p. This the same as the negation as failure rule used in Prolog for handling all negative goals.

Whether this is useful will depend on whether the goal $\sim p(t_1, ..., t_n)$ can be executed more or less efficiently than the original goal. Similarly if ~p is the original goal then p could be checked and ~p failed if p succeeds. The goal !p is more straightforward as it should succeed if either of p or ~p succeeds.

If the literals in the body of a clause are allowed to range over the other operators +, ¬, and ? then many more possibilities are raised. Basicly these operators require an NAF rule for their execution. For example ¬p should succeed if p evaluates to **u** or **f**. This can be determined by checking that p fails. That is ¬p has the same execution rule as not in Prolog. However, ¬p can also be proven if ~p succeeds. This is justified by the tautology ¬p←~p and the earlier resolution theorem (see Figure 3.). So one execution strategy would be to first execute ~p, if this succeeds then so does ¬p and nothing further need be done. If ~p fails then it is necessary to further check p to see if it succeeds. Similarly +p should succeed if ~p fails and can first be checked by seeing if p succeeds. Finally the goal ?p succeeds only if both p and ~p fail. Figure 4 summarizes the various execution strategies which can be used. Of course the usual care must be exercised when using NAF. Lloyd [1984, Chapter 3] gives a good explanation of when it can and cannot be used correctly.

VI SATISFIABLE PROGRAMS

One of the major differences between the approach to negation used here and the standard approach is that it is possible for programs to be unsatisfiable and hence useless. The theoretical foundations for the approach and the execution strategies explored above both require that the program be satisfiable. This is very similar to the idea of integrity constraints in data bases which are usually expressed as a set of conditions which must hold in any possible form of the data-base. The basic constraint that nothing can be simultaneously asserted true and false is a very convenient way of expressing such integrity

constraints. For not only does it provide a way of checking the validity of the database it also enables the 'constraints' to be used for doing useful computation during program execution.

There does remain the problem of actually checking when a program is satisfiable. One way to do this is to attempt the execution of the goal $\leftarrow p(x_1, ..., x_n) \wedge \sim p(x_1, ..., x_n)$ for each predicate p in the program **P**. If each such goal finitely fails then **P** must be satisfiable. This procedure in effect executes the clauses in $\mathbb{C}(\mathbf{P})$. Unfortunately the procedure does not always terminate and then other more general program proof procedures must be resorted to. Consider for example the following program:

$p(f(x)) \leftarrow p(x)$
$p(a) \leftarrow$
$\sim p(f(x)) \leftarrow \sim p(x)$
$\sim p(b) \leftarrow$

This has a model $\{p(a), p(f(a)), ..., \sim p(b), \sim p(f(b)), ...\}$ but the goal

$\leftarrow p(x) \wedge \sim p(x)$

will never terminate.

VII OTHER 3-VALUED LOGICS

Parts of the multivalued logic used here were originally described by Kleene[1938] who had the same interpretation of the truth values as provable, falsifiable and unknown. Lukasiewicz[1920], Bochvar[1939] and Slupecki[1946] described other fragments of the logic. Rescher[1969] provides an accessible description of all these works. Figure 5 lists various operators used here and the equivalent operators and languages using Rescher's [1969] notation. The propositional part of the logic has been used by Cleary[1987] to solve some problems in knowledge acquisition for expert systems.

A three valued logic with some of the operators used here was used by Fitting [1985] to produce a semantics for standard logic programming. His techniques applied in the current

context would lead to a four-valued logic. The only other use of multi-valued logics of which I am aware in the context of logic programming are the real-valued logics used by Shapiro [1983] and van Emden [1986] for uncertain reasoning, although in both these cases only the Horn clause fragment of the logic is described.

The techniques used here would seem to be applicable to many multi-valued logics. The major requirement is that one of the truth values not be expressible in the head of a clause, and then anything not otherwise forced can be assigned to this 'left-over' truth value. Cleary[1982] shows that there is an analogue to resolution in any multi-valued logic. So, it may be that the results of this paper can be extended to any such logic. This possibility is currently being investigated.

ACKNOWLEDGEMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

Bochvar, D.A. (1939) "On a 3-valued Logical Calculus and its Application to the Analysis of Contradictions," (in Polish) *Matematiceskij sbornik,* **4**, 165-166.

Cleary, J.G. (1972) "Resolution Theorem Proving in Multi-valued Logics," M.Sc. thesis, Dept. Mathematics, University of Canterbury, Christchurch, New Zealand.

Cleary, J.G. (1987 - in press) "Acquisition of Uncertain Rules in a Probabilistic Logic," *Int. J. Man-Machine Studies,* also in Proc. Knowledge Acquisition for Knowledge Based Systems Workshop, Banff, Canada, November, 1986.

van Emden, M.H. (1986) "Quantitative Deduction and its Fixpoint Theory," *J. Logic Programming,* **3**(1), 37-53.

Fitting, M. (1985) "A Kripke-Kleene Semantics for Logic Programs," *J. Logic Programming,* **2**(4), 295-312.

Flannagan, T. (1986) "The Consistency of Negation as Failure," *J. Logic Programming,* **3**(2), 93-114.

Kleene, S.C. (1938) "On a Notation for Ordinal Numbers," *J. Symbolic Logic* , **3**, 150-155.

Lloyd, J.W. (1984) *Foundations of Logic Programming* . New York, Springer.

Lukasiewicz, J. (1920) "On 3-valued Logic," *Ruch Filozoficzny,* 5, 169-171. Translated in McCall, S. ed., *Polish Logic: 1920-1939,* Oxford (1967), 16-18.

Rescher, N. (1969) *Many-Valued Logic.* New York, McGraw-Hill.

Shapiro, E.Y. (1983) "Logic Programs with Uncertainties," in *Proc. 8th I.J.C.A.I.* Bundy, A. (Ed). 529-532. William Kaufman.

Slupecki, N. (1946) "The Complete Three-valued Propositional Calculus," (in Polish) *Annales Universitatis Mariae Curie-Sklodowska,* **1**, 193-209.

Truth Table **t u f**	Operator Name	Logically Equivalent Expressions
f f f	f	
f u f		~!A A∧~A
f u t	~A	
f f t		¬¬~A ~¬~A
f t f	?A	¬A∧¬~A ¬!A ¬¬?A ~¬?A
f t t	¬A	~¬¬A
t f f		¬¬A ~¬A
t u f	A	~~A
t f t		~?A ¬?A
t u t	!A	A∨~A
t t f	+A	¬~A
t t t	t	

Figure 1. Logical and Procedural Descriptions of Unary Operators.

Original clausal form	Weak equivalence used	Final clausal form
$\mathbf{L} \vee A$	$\mathbf{A} \leftrightarrow \neg\neg A$	$\mathbf{L} \leftarrow \neg\mathbf{A}$
$\mathbf{L} \vee {\sim}A$	${\sim}\mathbf{A} \leftrightarrow \neg +A$	$\mathbf{L} \leftarrow +\mathbf{A}$
$\mathbf{L} \vee \neg A$		$\mathbf{L} \leftarrow \mathbf{A}$
$\mathbf{L} \vee ?A$	$?\mathbf{A} \leftrightarrow \neg !A$	$\mathbf{L} \leftarrow !\mathbf{A}$
$\mathbf{L} \vee !A$	$!\mathbf{A} \leftrightarrow \neg ?\mathbf{A}$	$\mathbf{L} \leftarrow ?\mathbf{A}$
$\mathbf{L} \vee +A$	$+\mathbf{A} \leftrightarrow \neg{\sim}\mathbf{A}$	$\mathbf{L} \leftarrow {\sim}\mathbf{A}$
$\mathbf{L} \vee {\sim}!A$	${\sim}!\mathbf{A} \leftrightarrow \neg\mathbf{f}$	$\mathbf{L} \leftarrow$
$\mathbf{L} \vee \neg +A$		$\mathbf{L} \leftarrow +\mathbf{A}$
$\mathbf{L} \vee \neg\neg A$		$\mathbf{L} \leftarrow \neg\mathbf{A}$
$\mathbf{L} \vee \neg ?A$		$\mathbf{L} \leftarrow ?\mathbf{A}$

Figure 2. Standard form for program clauses.

Classical resolution

Multivalued resolution

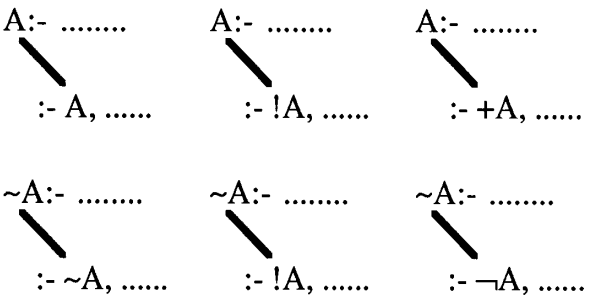

Figure 3. Allowed resolution steps.

Goal	Basic Rule	Alternate possible rule
p	execute p	execute ~p if it succeeds fail the original goal else execute p
~p	execute ~p	execute p if it succeeds fail the original goal else execute ~p
!p	execute p or ~p	
¬p	execute p, succeed if it fails	execute ~p succeed if it succeeds else execute p and succeed if it fails
+p	execute ~p, succeed if it fails	execute p succeed if it succeeds else execute ~p and succeed if it fails
?p	execute p and ~p, succeed if they both fail	

Figure 4. Possible execution strategies.

Operator used in this paper	Operator and language(s) using notation of Rescher[1969]
$\leftarrow$	$\leftarrow$ Slupecki [1946]
$\Leftarrow$	$\leftarrow$ $\mathbf{K}_3$
$\leftrightarrow$	$\leftrightarrow$ $\mathbf{B}_3{}^{E}$
$\wedge$	$\wedge$ $\mathbf{Ł}_3$ $\mathbf{K}_3$
$\vee$	$\vee$ $\mathbf{Ł}_3$ $\mathbf{K}_3$
$\neg$	$\neg$ $\mathbf{B}_3{}^{E}$ $\mathbf{K}_3$
$\sim$	$\neg$ $\mathbf{Ł}_3$ $\mathbf{B}_3$
+	W $\mathbf{Ł}_3{}^{W}$
	$\Diamond$ $\mathbf{Ł}_3$
?	† [Rescher, 1969, p57]
$\neg\neg\sim$	$\neg$ $\mathbf{Ł}_3{}^{W}$
$\neg\neg$	$\Box$ $\mathbf{Ł}_3$

Figure 5. Summary of equivalent operators in other 3-valued logics.