# Logic Programs With Uncertainties: A Tool for Implementing Rule-Based Systems

Ehud Y. Shapiro[1]
Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot 76100, ISRAEL

## 1. Introduction

One natural way to implement rule-based expert systems is via logic programs. The rules in such systems are usually definite clauses, or can easily be expressed as such, and the inference mechanbms used by such systems are built into the Prolog interpreter, or can be implemented in Prolog without much effort.

The one component of expert systems which is not readily available in logic programs is a language for specifying certainties of rules and data, and a mechanism for computing certainties of conclusions, given certainties of the premises. Clark and McCabe [1] suggest an implementation technique for solving this problem. They augment each predicate in the rule-language with an additional argument, whose value is the certainty of the solution returned in this predicate, and augment the condition of each clause with an additional goal, whose purpose is'to compute the certainty of the conclusion of the clause, given the certainties of solutions to the goals in the condition of the clause.

In this paper we propose a different way of implementing rule-based expert systems within Prolog, in which evaluation of certainties of solutions is carried out at the meta-level, within the logic program interpreter itself. This resulting framework, called *logic programs with uncertainties,* has the following properties:

- It is amenable to theoretical analysis. In particular, a precise semantics can be given to logic programs with uncertainties.
- Standard logic programs are a special case of logic programs with uncertainties. If all certainty factors are 1, then the semantics defined and the interpreters developed degenerate to the standard semantics and the standard interpreter for logic programs.
- Since the semantics of logic programs with uncertainties is simple, it is easy to apply the debugging algorithms developed in [3].

We consider the last point to be of great importance. Algorithmic debugging can provide the essential link between the expert and the executable realization of his knowledge, and thus facilitate the process of knowledge transfer and debugging.

The paper defines the syntax and semantics of logic programs with uncertainties, develops an interpreter for such programs, and suggest how algorithmic debugging can be used by an expert to debug such rule-based systems. Examples are not included due to space limitations.

## 2. Logic programs with uncertainties

A *definite clause* is a clause of the form A«— *B,* where *A* is an atom and *B* is a conjunction of zero or more atoms. A *certainty factor c* is a real number, greater than zero and less than or equal to one. A *certainty function f* is a function from multisets of certainty factors to certainty factors. A *logic program with uncertainties* P is a finite set of pairs <A*-J3,/>, where *A*-B* is a definite clause and / is a certainty function.

The certainty function is used to compute the certainty of the conclusion of a clause, given the multiset of certainties of solutions to goals in the condition of the clause. We require from a certainty function / that for every multiset S, f(SU{1})=/(£)» and that / be monotonic increasing, which means that *S<S'* implies f(S)<f('S'')» where < is the partial order over multisets, defined as follows. Let *S* and *X*={x₁* x₂,..., *xₙ},* be two multisets, n>0. Then *X<S* iff there is a multiset *Y={Y₁ y₂* •••» yₙ} such that *ScY* and x₁<•/,-, 0<i<n. Our theoretical treatment is independent of the particular certainty functions chosen, as long as they satisfy these two requirements.

'This research was carried while the author was at Yale University, supported by the National Science Foundation, grant No. MCS8002447.

For every pair $<A \leftarrow B, f>$ in $P$, the number $f(\{\})$ is called the *certainty factor of* $A \leftarrow B$ *in* $P$.

We define semantics for logic programs with uncertainties. An *interpretation* $M$ of a logic program with uncertainties $P$ is a set of pairs $<A,c>$, where $A$ is a ground atom and $c$ is a certainty factor. We require that $M$ contains at most one pair $<A,c>$ for any atom $A$.

A ground atom $A$ is *true in* $M$ *with certainty* $c$ iff there is a pair $<A,c'>$ in $M$ such that $c \leq c'$. An atom $A$ is *true in* $M$ *with certainty* $c$ iff for every ground instance $A'$ of $A$, $A'$ is true in $M$ with certainty $c$ (instances are taken over the Herbrand base of the program $P$).

**Definition 2.1:** Let $A \leftarrow B_1, B_2, ..., B_n$, $n \geq 0$ be a ground definite clause, $f$ a certainty function and $S$ the multiset of certainties $\{c_1, c_2, ..., c_n\}$, such that $<B_i,c_i>$ is in $M$, for every $1 \leq i \leq n$ (if no such pair exists for some atom $B_i$ then $S$ is considered undefined). Then $A \leftarrow B$ is *true in* $M$ *with respect to* $f$ iff either $S$ is undefined or $A$ is true in $M$ with certainty $f(S)$.

A definite clause $A \leftarrow B$ is *true in* $M$ *with respect to* $f$ iff any ground instance $A' \leftarrow B'$ of it is true in $M$ with respect to $f$.

A logic program with uncertainties $P$ is *true in* $M$ iff for any pair $<A \leftarrow B, f>$ in $P$, $A \leftarrow B$ is true in $M$ with respect to $f$.

Note that if the certainty factors of all clauses in $P$ are 1, the semantics thus defined is equivalent to the standard semantics of logic programs [2].

We define a partial order on interpretations. Our goal is to define the analog of the minimal model of standard logic programs. Let $M$ and $M'$ be interpretations. Then $M \leq M'$ iff for any pair $<A,c>$ in $M$ there is a pair $<A,c'>$ in $M'$ such that $c \leq c'$.

**Definition 2.2:** Let $P$ be a logic program with uncertainties. Then $M(P)$ is defined to be the minimal interpretation $M$ under $\leq$ such that $P$ is true in $M$.

To see that a minimal interpretation exists, one can define intersection of interpretations in the natural way, and verify that for any two interpretations $M$ and $M'$, if $P$ is true in $M$ and $M'$ than $P$ is true in $M \cap M'$, and that $M \cap M' \leq M$. It is easy to see that $M(P)$ is the intersection of all interpretations $M$ such that $P$ is true in $M$.

A less trivial exercise is to associate a transformation $\tau_P$ with the program $P$, and to prove that the least fixpoint of $\tau_P$ is $M(P)$.

This completes the definition of the semantics of logic programs with uncertainties.

## 3. Interpreters for logic programs with uncertainties

We describe interpreters for logic programs with uncertainties, written as a logic program. First we show a standard meta-circular interpreter for logic programs. We use upper-case strings as variable symbols, and lower-case strings for all other symbols. We assume that the conjunction $B_1, B_2, ..., B_n$ is represented as $(B_1,(B_2,(...,(B_n,true)...)))$, that the atomic clause $A$ is represented as $A \leftarrow true$, and that a program $P$ is represented as set of clauses *clause*$((A \leftarrow B))$ for any clause $A \leftarrow B$ in $P$. The semantics of *solve*$(A)$, given $P$, is "$A$ is provable from $P$"

```
solve(true).
solve((A,B)) ← solve(A), solve(B).
solve(A) ← clause((A←B)), solve(B).
```

**Figure 1:** A meta-circular interpreter for logic programs

An interpreter for logic programs with uncertainties is an extension to this interpreter. It assumes that the program $P$ is represented as clauses "*clause*$(<A \leftarrow B,F>) \leftarrow$" for any clause $A \leftarrow B$ with certainty function $F$ in $P$. Also, multisets are represented using Prolog lists: [] is the empty list, which represents the empty set, and [X|Y] is the list whose head is $X$ and tail is $Y$, which represents the multiset $\{X\} \cup Y$.

```
solve(true,[]).
solve((A,B),[X|Y]) ← solve(A,X), solve(B,Y).
solve(A,F(S)) ← clause(<A←B,F>), solve(B,S).
```

**Figure 2:** An interpreter for logic programs with uncertainties

The semantics of *solve*$(A,C)$ is "$A$ is provable from $P$ with certainty $C$". The interpreter returns in $C$ an unevaluated expression with occurances of certainty function symbols in it. To compute the actual certainty factor one has to evaluate this expression. Note that, due to the (nonlogical) way the certainty of solutions is computed, the second argument of *solve* can be used for output only, or, in other words, the interpreter is applicable only in case the certainty factor is not instantiated in the input goal. Also, most Prolog implementations do not allow expressions of the form F(S), where *F* is a variable symbol. There are standard techniques to get around the problem.

Ideally, one would like to specify some certainty threshold, with the intention that the interpreter will compute only solutions with certainty above this threshold. We develop a special purpose pruning interpreter for a particular scheme of certainty functions, and then show how to generalize it.

Since the semantics defined for logic programs with uncertainties proves to be insensitive to the particular certainty functions chosen, we tend to believe that no theoretical argument will be able to decide between the reasonable alternatives. Sociologically speaking, this conjecture has proved itself so far. Each school of expert systems is using its own particular way of computing certainties, with no noticeable difference in the validity of their results.

So we pick our pick. The certainty functions we choose, *cf*(5), computes the product of *c* and the minimal element of 5, in case *S* is not empty, and returns c otherwise. These functions meets the requirements stated above. As the difference between the functions associated with the different clauses is just the constant c, we represent clauses as pairs *<A<-B,c>,* where *c* is the certainty factor of *A+-B* in *P.* The interpreter in Figure 3 below has this function scheme built-in. This interpreter receives as input a goal and a threshold of required certainty for solutions, and prunes computations for which it is evident that any solution found along them will not meet this threshold. The semantics of *8olvt(A,T,C)* is "*A* is provable from *P* with certainty C,and *C>r.*

The way the interpreter prunes low-certainty execution paths is via the call times*(7",F,7). If the results of dividing *T* by *F* is greater than one, the call fails, as 7" is not a certainty factor then. Note that *times* serves three functions in the interpreter: it computes multiplication and division, and prunes computation paths which fail to pass the desired certainty threshold. A similar interpreter can be built for most reasonable schemes of certainty functions.

This interpreter has the following interesting property: it always terminate if every clause in the program it executes has certainty factor less than 1.

```
solve(true,T,1).
solve((A,B),T,C) ←
        solve(A,T,X), solve(B,T,Y), min(X,Y,C).
solve(A,T,C) ←
        clause(<A←B,F>), times(T',F,T),
        solve(B,T',C'), times(C',F,C).

times(X,Y,Z) ← "X times Y is Z,
        and they all are certainty factors
        (i.e. X, Y and Z are all less than or equal to 1
        and greater than 0)".

min(X,Y,Z) ← "Z is the minimum of X and Y".
```

**Figure 3:** A specialized interpreter that accepts a certainty threshold

program it executes has certainty factor less than 1. More precisely, if the largest certainty factor of any clause in *P* is $c^*$, $c^* < 1$, and the interpreter is invoked with a certainty threshold *t*, $0 < t \leq 1$, then the number of goal reductions on any execution path of this interpreter would be no more than the the smallest positive integer *n* for which $c^{*n} < t$.

A similar property holds for the following, more general interpreter, shown in 4. The semantics of *solve(A,T,F,C)* is "*A* is provable from *P* with certainty *C*, and $F(C) \geq T$". The variable *T* carries the input certainty threshold, *F* the expression representing the certainty function computed so far, and *C* the certainty of the solution to *A*, which is a certainty factor in case *A* is an atomic goal, and a multiset of certainty factors in case *A* is a conjunctive goal.

```
solve(true,T,F,[]).
solve((A,B),T,F,[X|Y]) ←
        solve(A,T,λS(F(S)),X),
        solve(B,T,λS(F([X|S])),Y).
solve(A,T,F,F'(C)) ←
        clause(<A←B,F'>),
        F(F'([]))≥T,
        solve(B,T,λS(F(F'(S)));C).
```

**Figure 4:** An abstract interpreter with certainty threshold

The way the interpreter trims low certainty computation paths is via the goal $F(F'([])) \leq T$. The maximum of any certainty function *F* is at [], by the monotonicity requirement. Hence if at its maximum the value of the function $\lambda X(F(F'(X)))$ is still less than the certainty threshold F, the computation is hound not to meet this threshold.

## 4. A Note on debugging logic programs with uncertainties

Since a logic program with uncertainties has semantics, one can debug it like any other program, if one knows its intended interpretation. That is, if one knows the input-output relations it is supposed to compute, and their associated certainties. In particular, the diagnosis algorithms described in [3] are applicable in such a situation.

Assume that a logic program .with uncertainties has a conclusion A whose computed is judged by the expert to be too high. We can conclude that, according to the interpretation the expert has in mind, the program contains at least one false clause. Such a clause can be detected by querying the expert about the truth (or certainty) of intermediate conclusions obtained during the proof of A, as done in [3].

The advantage of having such a diagnosis algorithm is more valuable for the debugging the rule-base of a system by an expert then for normal program debugging: In normal program debugging the debugger is a programmer, who not only knows the intended declarative semantics of the program (i.e. what it should compute), but its procedural semantics (i.e. how it computes) as well.

This is not necessarily the case with experts and rule-based systems. One reason for expert systems being composed of a rule-base and an inference mechanism is to allow the expert to effectively transfer her knowledge in a declarative form to the system, even when she is ignorant of the particular inference mechanism used. The debugging algorithms allow the expert to maintain this appropriate ignorance even when she is debugging the rules she suggested, since these algorithms simulate the execution of the inference mechanism, and query the expert for declarative information only.

## 6. Conclusions

Logic programs with uncertainties provide a sound theoretical basis for systems which contain only an approximate description of a domain. Their clear declarative semantics allows the expert to tune the system to reflect his knowledge, while maintaining ignorance of the inference mechanism the system implements.

## References

[1] K. L. Clark and F. G. McCabe.
Prolog: a language for implementing expert systems.
In D. Michie and Y. H. Pao (editors), Machine Intelligence 10, . , , 1982.

[2] M. H. van Emden and R. A. Kowalski.
The semantics of predicate logic as a programming language.
Journal of the ACM 23:733-742, October, 1976.

[3] Ehud Y. Shapiro.
ACM Distinguished Dissertations Series: Algorithmic Program Debugging.
The MIT Press, 1983 (in press).