Uncertainty Reasoning in Prolog with Layered Meta-Interpreters

L. Ümit Yalçınalp and Leon Sterling
Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio, 44106, USA.
umit@bach.ces.cwru.edu, leon@alpha.ces.cwru.edu

Abstract

This paper proposes an architecture for uncertainty reasoning for rule based systems in Prolog. Previous work has assumed predefined calculus for the propagation of uncertainty restricted to a single line of reasoning or where no negation is allowed. We identify the issues that need to be considered for both handling negation and taking a more global view. We show that a layered meta-interpreter is a good flexible architecture that can accommodate different sets of assumptions for incorporating uncertainty in rule based systems in Prolog.

AI Topic: Uncertainty Reasoning, Knowledge Engineering
Domain Area: Meta-Programming, Expert System Shells
Language: Prolog
Status: Ongoing
Effort: 3 months one-person
Impact: Development of flexible expert system shells which can be tailored to accommodate different calculi for representing uncertainty.

1 Introduction

There are many models and methods in dealing with uncertainty, beliefs and evidential reasoning in knowledge based systems. The methods depend on the representation of uncertainty within the language and the uncertainty reasoning calculus which describes its semantics. The methods used in reasoning with uncertainty and defining its calculus are divided into three main groups, namely using ad-hoc methods, using point probabilities based on Bayesian models and using multi-valued probabilities based on non-Bayesian models. The primary example for employing ad-hoc reasoning is MYCIN [2]. The primary example for using Bayesian models is using Dempster-Shafer theory for evidential reasoning and probabilistic logic. A good source which comprises these models and discusses their limitations is by Pearl [6].

Methods for incorporating uncertainty within logic programming have also been proposed. The representation of uncertainty is achieved by augmenting the representation of each clause by a suitable representation of certainty which can be single or multi-valued. The first attempt incorporated uncertainty reasoning by annotating each clause by a single value (Clark and McCabe [3]). This value indicates the certainty value associated with the solution of the clause if the conditions (premises) of a clause are solvable. However, an additional goal for the calculation of uncertainty has to be given within the definition of each clause in [3]. This is due to the attempt to integrate the calculus within the language.

It is well known to use Prolog meta-interpreters for expert system shells for controlling the computation and making selected properties of the deduction explicit. Shapiro [8], separated the calculation and propagation of uncertainty via a meta-interpreter. The major limitation of this work is the inability to combine multiple lines of reasoning that indicate multiple evidences for obtaining the solution of a goal. Examples for integrating this simple form of uncertainties within Prolog computations are given in the text [10].

Various ad-hoc uncertainty methods are implemented in Prolog. For example, Prolog versions of MYCIN floating around in the logic programming community inherit MYCIN’s uncertainty calculus. There have been also frame-based uncertainty reasoning schemes, such as [12].

Integration of uncertainty reasoning with Dempster-Shafer theory with Prolog is proposed by Baldwin [1] based on the open world assumption. Each clause p is augmented by a two-valued certainty factor \([\text{Bel}(p), \text{Pl}(p)]\). \(\text{Bel}(p)\) indicates the belief that \(p\) is true, where \(\text{Pl}(p)\) indicates the plausibility of the predicate, or the degree of belief that it can not be refuted. Plausibility is defined as \(\text{Pl}(p) = 1 - \text{Bel}(\neg p)\). Baldwin’s work explicitly uses Dempster-Shafer theory for propagating beliefs, and discusses how to combine uncertainties in clauses that are not mutually exclusive by using Dempster’s rule of combination.

Yet another approach is the recent ongoing work of Ng and Subrahmanian [5] with annotated two-valued logics. The probability pair value attached to clauses...
shows the range in which a probability of an event must lie. Their intent is to define a language which is totally faithful to probabilistic logic and describe its model theory. Their work is the first to give a probabilistic semantics for expressing quantification in logic programming.

Our intent in this paper is not describing another logic for integrating uncertainty within logic programming. We take a pragmatic approach and argue that reasoning with uncertainty can be achieved by a flexible meta-architecture that is suitable for describing and reasoning with different representations of uncertainties, combining multiple lines of reasoning and reasoning with unknowns and negation. Shocken and Finin [7] propose a similar approach and use a meta-interpreter for describing ad-hoc and bayesian models. Shekhar and Ramamoorthy provide a meta-interpreter along the lines of [1]. We propose a layered meta-interpreter in Prolog that is based on [11] and can represent multiple lines of reasoning. We claim that it provides a clean framework which subsumes the earlier work based on meta-interpreters in [8, 10, 1, 7].

2 Issues for Uncertainty Reasoning

In this section, we discuss some of the issues that should be addressed for practical uncertainty reasoning with Prolog\(^1\). We discuss our assumptions in Section 3.

In this paper, we assume that the representation of a Prolog clause is augmented by a certainty factor which indicates a conditional certainty of the head of the clause when the body of the clause is true. For example, the clause \( A \leftarrow c_{i}(C_{F}), B_{i}, \ldots, B_{n} \), the certainty factor is given by \( c_{i}(C_{F}) \) of \( A \) when the body of the clause is true. The clauses without certainty factors correspond to clauses with the maximum certainty factor, i.e. 1, or [1, 1].

We identify the following issues for calculating the certainty of a predicate.

1. Representing and Combining Multiple Evidence: The result of a successful Prolog deduction simply involves one branch in the proof tree. However, the other branches which provide the same answer substitution in the search tree correspond to multiple evidence that indicate the particular answer and they should be accounted for cumulatively. Operationally, this is not possible by calculating uncertainties based on a single line of reasoning as in [8, 10]. Therefore, the architecture should provide multiple lines of reasoning to account for multiple evidence.

   There are different calculi for assigning a certainty value to combine multiple evidence [7]. In [4], the maximum of the pooled certainties from multiple evidence is assumed. For multi-valued uncertainties, different intervals for multiple evidences are also used for resolving conflicts in combining evidence as in [1].

   Operationally, getting all the lines of computation and their respective certainty values can be obtained with a set predicate, such as findall. This approach is taken in the two current studies of handling uncertainty by using meta-interpreters [9, 7]. In the former, multiple evidence is considered only for the top level query. However, this approach should be integrated at different levels of the computation, by considering different lines of reasoning to compute a composite uncertainty for all the rules in the computation. This approach is taken in [7]. However, multiple evidence are not grouped together for each answer substitution. Therefore, their approach is suitable for only ground queries.

2. Combining Answer Substitutions: The second issue concerns reconciling variable bindings. For example, suppose a predicate \( p/1 \) has two different solutions, \( p(1) \) with certainty \( p_{1} \) and \( p(2) \) with certainty \( p_{2} \). If we pose the query \( p(X) \), it is not clear what the certainty of the answer should be. Since we can not expect the certainty to resemble a distribution function, every ground answer should be handled separately.

3. Handling Negation: Negation for Prolog programs is based on the negation-as-failure rule. Operationally, dealing with negation requires the computation of all lines of reasoning for a goal. All possible lines of reasoning for a negated goal must be obtained in order to show that all of them result in failure [11].

Handling negation for reasoning with uncertainty requires a similar but a revised approach. We need to consider two issues which are closely related.

a. Handling the missing facts or unknowns in the computation is an important issue. For example, for a program with facts \( p(a) \) and \( p(b) \), what can we say about the certainty involved with \( \neg p(c) \)? Our assumptions about the unknown facts naturally shapes how we deal with negation. Single valued probabilities do not allow a distinction between false and unknown values. When there is neither evidence to support or refute a fact, 0 can be assigned conservatively by using a point valued certainty. A multi-valued representation, on the other hand, distinguishes between true, false, and unknowns explicitly. A [0, 1] pair to indicate full unsuresnness in the proposition indicated by the missing rule or fact. Rules/facts that are definitely known to be false are represented by [0, 0] or known to be true by [1, 1] explicitly with multi-valued certainties as in the references [5, 1].

b. In the context of reasoning with uncertainty, finite failure of all the branches of the search tree for a negated predicate is meaningless. However, combining uncertainties for all lines of reasoning giving the same answer substitution for the negated predicate is necessary, including the unknown facts. After combining the multiple evidence, the certainty for the negated predicate is determined by the rules of the uncertainty calculus. For example, for single valued uncertainties, a certainty of not \( p \) can be obtained after a composite certainty for \( p \) acquired, such as \( \text{prob}(\neg p) = 1 - \text{prob}(p) \). For multi-valued certainty \([\alpha, \beta]\), then its negation, \( \neg p \) simply has the certainty in the interval \([1 - \beta, 1 - \alpha]\) in [1].

The earlier approaches [8, 10], do not deal with nega-

\(^1\)In our discussion, we will use the term certainty of a predicate somewhat freely to illustrate the degree of belief that a preposition represented by a predicate is true.
tions correctly, since there is no mechanism to incorporate multiple evidence. Later approaches to uncertainty reasoning in [9, 7] do not address negation.

3 Layered Meta-Interpreters and Uncertainty

As shown in [7], the inference which is capable of handling multiple lines of reasoning can be represented independently of the uncertainty calculus. A meta-interpreter can be used for this purpose as a meta-meta specification which can be tailored for the specific application in hand by specifying the uncertainty calculus. It is important to integrate handling unknown facts and negation with multiple lines of reasoning. In this section, we present a layered Prolog meta-interpreter as an alternative architecture for providing a unified framework.

We have previously shown the inadequacy of the extensions to the well-known vanilla meta-interpreter as basis for expert systems shells and also illustrated the utility of a fail-safe layered meta-interpreter for incorporating negation which requires handling multiple lines of computation and missing facts in [11]. This basic scheme is especially useful to generate proofs for both success and failure within one computation within the context of explanation [11].

The enhancement of the layered architecture for uncertainty reasoning is as follows. One layer, solve_bottom, traverses each branch in the search tree, and the other layer, solve_top, realizes Prolog's behaviour by successively generating each branch. The function of solve_top in [11] is to search for a successful computation by monitoring the state of the computation and if the computation results to initiate back-tracking when a failure occurs. In contrast, our goal in a computation that involves certainties is to compute a certainty value considering multiple evidence, hence combining all lines of reasoning. The function of the bottom layer is to generate a single line of reasoning and calculate its certainty as well as composition of conjunctions, disjunctions, and the top layer monitors the computation as well as computing the composite certainty from different lines of reasoning. In the basic layered interpreter [11], our emphasis was to use all possible branches of the search tree to represent failure explicitly. In this extension, our focus is to combine all lines of reasoning. The computation proceeds in two layers until all the possible branches of reasoning are exhausted.

The layered interpreter is given in Figure 1. This interpreter also computes a proof of the deduction. We use the predicate clause_new to obtain each rule in the assumed format as described in Section 2. The filter_uncertainty predicate in the top layer handles the certainty calculations for conjunctions, disjunctions and negations and initiates the computation of all possible branches. The predicate store_proof is used to store the branches of the proof temporarily based on the Name which uniquely identifies a goal. They are used at the end of the computation along with the computed certainty values attached to each branch. The computation terminates at the top level when the first clause of solve_top ultimately fails. At the end of the computation for each predicate, all the certainty values attached to different lines of reasoning are extracted by the combine_prob predicate at the top layer.

A careful reader will observe that we require both of the conjuncts in a conjunction to be computed at the top level as illustrated by solve_bottom_and in Figure 1. Recall that we made the assumption to provide each solution one by one at the top level, where each answer substitution is given with its composite certainty. This should be taken into account in calculating the certainty of a conjunct, by using solve_top for each conjunct in a conjunction.

In computing a conjunction, a threshold can be used, by defining continue_conj/1 appropriately, to indicate whether the second conjunct should be solved. Unknown facts are represented explicitly at the bottom layer of the interpreter as illustrated in Figure 1. The COOP shell [9] represents unknowns by adding an auxiliary clause in the knowledge base. This method
requires updating the knowledge base and is not necessary as illustrated here by using the appropriate representation within the meta-interpreter.

Posing negated nonground goals as queries, such as not p(X)?, is similar to using use of negation in Prolog. By using the meta-interpreter, such uses can be reported and prevented.

The programmable sections of this interpreter are the filter:failure, combine:prob, truthval, falseval and unknownval which don't affect the nature of the inference. They are used to define the uncertainty reasoning calculus. The function of combine:prob is

- to group the solutions with respect to different answer substitutions. For example, a query p(X), when there are multiple solutions, i.e. p(1), p(2), ..., we choose to provide each answer substitution with the composite certainty corresponding to that substitution. The reasons for choosing this approach is obvious since we would like to obtain answer substitutions from our queries by using a Prolog program. However, knowing and grouping the answers and their corresponding proofs in advance is not possible. Therefore, all the possible answers are computed for a predicate and then grouped together by combine:prob.

- to eliminate the branches that do not support an answer, i.e. branches with 0 or [0,1] certainty from calculations of uncertainty.

- to combine the probabilities for each answer substitution from different lines of reasoning. This is provided by a combine predicate.

- to return each solution in order as requested, i.e. p(1), then p(2), ....

The general structure of filter:uncertainty is given below. We provide an example calculus for describing uncertainty by using the functions neg, and and imp which describe negation, conjunction and implication respectively. Uncertainty is also easily described based on Dempster-Shafer theory as in [1] by changing the functions in filtering, the truth intervals and combine accordingly.

```
filter:uncertainty(Val, fact, Val, fact) ← !.
filter:uncertainty(U, Name, _, sys(_G)) ← !.
filter:uncertainty(F, Name, _, Proof) ←
    (falseval(F); unknownval(F)),
    store:proof(Name, Proof,F), fail.
filtered:uncertainty(invert(Cf),_,U,Proof) ← !,
    neg(U,Cf).
filtered:uncertainty(and(U1,U2),_,U,Proof) ← !,
    and(U,U1,U2).
filtered:uncertainty(body(CBody,Cl),_,Name,_,Proof) ←
    imp(U,CBody,Cl),
    store:proof(Name, Proof,U), fail.
```

4 Examples

We give a simple example to illustrate different approaches that can be achieved by using the layered interpreter.

```
p(X) ← cf(0.6), not q(X).
q(X) ← cf(0.7), r(X).
qu(X) ← cf(0.8), s(X).
```

r(a). r(b). s(a).

Combining two different independent solutions for the goal, p, can be defined in several ways. For example, taking the set union of the answers is defined as the first rule for combination

```
combine([U1,U2,U3], U) ← U is U1 + U2 + U3 - U1*U2 - U1*U3 - U2*U3 + U1*U2*U3.
```

or by using Maier and Warren's method of using the maximum certainty.

```
combine(List, Max) ← list(List).
map_list(List, List).
```

Based on these functions and the first rule for combination, the query p(a) is assigned 0.036 certainty. The certainty associated with q(a) is 0.94 combining both of the clauses that will give the solution q(a). The certainty factor for not p(a) is computed to be 1 - 0.94 = 0.06. For the subsequent queries, p(b) is assigned 0.18, and p(c) is assigned 0.6. Since there is no clause to prove q(c), the certainty associated with q(c) is 0. As a result, not q(c) is computed as 1 - 0 = 1, which propagates to assign the maximum certainty to the clause p(c).

5 Conclusions

The layered meta-interpreter provides several advantages in extending Prolog for uncertainty reasoning.

Our approach provides flexibility by combining multiple evidence leading to the same conclusion for uncertainty reasoning. One of the important features of our design is the ability to separate the inference and uncertainty reasoning by an appropriate meta-interpreter. In our example, we have incorporated uncertainty by using a single certainty value attached to each clause and defining uncertainty calculus by modifying the functions used by filter:uncertainty and combine:prob.

The same architecture can be tailored to accommodate different methods and representations for calculating uncertainty, i.e. a multi-valued certainty representation similar to [1]. In terms of the goals in approaching flexibility, our approach is similar to [7]. Instead
of using a set predicate to obtain all lines of reasoning, we obtain and reason with multiple evidences with the different granularity provided in layers. Unlike [7], this feature makes handling nonground queries possible. We used Prolog directly for specifying uncertainty calculus. In [7], a simple language which compiles into Prolog is provided for this purpose.

Having a layered approach not only provides a flexible architecture for rule based systems in reasoning with multiple evidence, but also incorporates handling negation and unknowns. These aspects are not present in the previous approaches based on inter-preters. The extended interpreter in Figure 1 handles negation by considering all lines of reasoning for a negated goal and by using axioms for calculating uncertainty for negation at the top layer. Furthermore, unknown facts are handled by the meta-interpreter without changing the knowledge, unlike [9]. A default certainty value is assigned to unknown facts by the fail-safe bottom layer where the unknowns are explicitly handled as one branch of reasoning. Alteration of the knowledge base is not required and the inference is defined independent of the domain.

We also achieve different levels of granularity. On one hand, we prune individual branches of the search tree at the bottom layer. On the other hand we access the branches together at the top layer to obtain a combined value of certainty. The certainty functions are assigned at the bottom level and calculated at the top level to incorporate different representations for uncertainty as well as managing proofs in the computation. We represent the proofs to provide an explanation for illustrating the uncertainty calculations for the rules. This is also an issue for resolving conflicts when multiple evidence is present and contradictory. [1] illustrates the use of Dempster's combination. However, when totally contradictory evidence exists, i.e. indicated by [0,0] and [1,1], an explanation might be provided to request user interaction. This feature can be integrated in the combine predicate within the definition of combine_prob.

Another issue in handling uncertainty is the dependencies in evidence. When the rule structure is not a tree [6], this means that the same evidence can be used more than once in the inference of a particular result. We have assumed the independence of evidence in our work for handling multi-valued uncertainties.

Our approach is faithful to Prolog's computation. It differs from [5] for probabilistic logic programming where a new proof theoretic semantics is defined. We take a pragmatic approach by extending the layered meta-interpreter and overcome the difficulties in the earlier meta-interpreter approaches for integrating uncertainty reasoning in Prolog successfully.

Acknowledgements

We thank Tim Finin for sending [7] which shaped our ideas, and our reviewers for corrections. This research was supported under NSF Grant No. CCR90-00387.

References