PROLOG META-INTERPRETERS FOR RULE-BASED INFERENCE UNDER UNCERTAINTY

by

Shimon Schocken[1]

Information Systems Area
Graduate School of Business Administration
New York University
90 Trinity Place
New York, N.Y. 10006

Tim Finin

Unisys Corp. and
Computer Science Department
Moore School of Engineering
University of Pennsylvania

September 1987
(Revised October 1987)

Center for Research on Information Systems
Information Systems Area
Graduate School of Business Administration
New York University

Working Paper Series

CRIS #165
GBA #87-91

---

[1]Please direct all inquiries and comments to Shimon Schocken, 624 Tisch Hall, 40 West 4th Street, New York, NY 10003, or by electronic mail to B20.s-schocken@kl.gba.nyu.edu

Abstract

Uncertain facts and inexact rules can be represented and processed in standard Prolog through meta-interpretation. This requires the specification of appropriate parsers and belief calculi. We present a meta-interpreter that takes a rule-based belief calculus as an external variable. The certainty-factors calculus and a heuristic Bayesian belief-update model are then implemented as stand-alone Prolog predicates. These, in turn, are bound to the meta-interpreter environment through second-order programming. The resulting system is a powerful experimental tool which enables inquiry into the impact of various designs of belief calculi on the external validity of expert systems. The paper also demonstrates the (well-known) role of Prolog meta-interpreters in building expert system shells.

1. Introduction

More than any other programming language, Prolog means different things to different people. In this paper, we focus on some aspects of Prolog which make it particularly useful for building systems for non-categorical rule-based inference. The basic notion of logic programs with uncertainties is due to a paper of this title by Shapiro (1983). The present paper elaborates on this concept in the context of expert systems and presents several extensions to the basic idea. The computational tools that resulted from this research turned to be very useful in experimentation with alternative techniques for rule-based inference under uncertainty.

In a logic program with uncertainty, rules and facts are parameterized by some sort of a "degree of belief." The program is designed to compute posterior beliefs in goals as a side-effect of standard theorem proving. Belief update can be performed either within the logic program itself (e.g. Clark & McCabe, 1982, Alvey et al, 1986), or at higher, meta-level of interpretation (Shapiro, ibid). A Meta-interpreter is an interpreter of a language written in the same language. In Prolog, meta-interpreters have proven to be particularly useful in building expert system shells. The basic idea is that Prolog is already a very capable first-order inference-engine; turning this raw power into a full-featured shell is basically a matter of adding functionalities to the standard language. For the sake of modularity, this is best accomplished by creating specialized

meta-interpreters and enhancing them incrementally (Sterling, 1986).

Prolog meta-interpreters were developed to add a number of essential capabilities found in most commercial expert system shells. For example, Hammond & Sergot (1984) extended the inference-engine with a "query the user" facility which obtains missing information through interactive consultation. Sterling and Lalee (1986) developed techniques to explain the system's line of reasoning. A number of authors, e.g. Dincbas (1984) and Pereira (1982), have shown how the fixed control structure of Prolog can be short-cut and modified to suit various inferential needs. Baldwin and Monk (1986) developed a meta-interpreter for inexact reasoning based on the Dempster-Shafer model (Shafer, 1976).

The motivation for this paper came from the first author's interest with experimenting with a variety of belief update models in expert systems. It soon became clear that such experiments require a computational environment which (a) simulates a standard rule-based inference algorithm, and, (b) allows a great deal of design flexibility with respect to creating and modifying alternative belief calculi. This need was satisfied effectively by extending work of the second author on Prolog meta-interpreters. In the process of developing these tools, we became aware of a recent paper by Sterling (1986) describing the analogy between Lisp Flavors and Prolog meta-

interpreters. Sterling's paper provided an elegant theoretical framework within which our work can be viewed as a special case of flavor mixing.

The paper begins with a synopsis of rule-based inference under uncertainty in the context of expert systems. Techniques for representing and parsing uncertain facts and inexact rules within Prolog are then described. Next, the building blocks of a belief update model are defined and implemented as logic programs. These individual modules, of which systems like MYCIN and PROSPECTOR are built, are then integrated into an overall meta-interpreter called SOLVE. The unique feature of SOLVE is that it takes a belief calculus as an external parameter. The paper proceeds to present Prolog predicates which implement the certainty-factors calculus and an ad-hoc Bayesian belief update model, and shows how these can be easily mixed with SOLVE's theory. The paper concludes with comments on the suitability of this environment to experimentation on the validity of rule-based inference in non-deterministic domains.

2. Rule-based inference under uncertainty

The mathematical and cognitive underpinnings of rule-based (production) systems are well known, and the reader is referred to Davis and King (1984) and Newell (1973) for elaborate discussions. Due to its proximity to first-order predicate calculus, the rational basis of categorical rule-based inference is normally unchallenged. This validity, however, does not

extend naturally to applications involving uncertain facts and heuristic inference rules. Under such conditions, a rule-based system may be viewed as a non-categorical classification program, designed to map a set of observed facts on a set of one or more explaining hypotheses (Cohen, 1985). This inexact matching algorithm is carried out by applying modus ponens repeatedly to a set of rules of the form <IF e THEN h Bel> (throughout the paper, e and h stand for a piece of evidence and an hypothesis, respectively). The postfix Bel is a degree of belief, which, broadly speaking, reflects an expert's confidence in the logical entailment associated with the implication e->h. The problem, simply put, is this: given the prior belief in h and all the degrees of belief that parameterize rules and facts that ultimately imply h, how does one compute the posterior belief in h? In expert systems, this is typically accomplished by some sort of a belief language.

According to Shafer and Tversky (1985), the building-blocks of a belief language are syntax, calculus, and semantics. In the context of rule-based inference, syntax corresponds to a set of degrees of belief which parameterize uncertain facts, inexact rules, and competing hypotheses. The degrees of belief associated with rules are elicited from a domain expert as the knowledge-base is being constructed. Factual degrees of belief are obtained interactively through consultation. Posterior degrees of belief are computed through a set of operators collectively known as a belief calculus. The semantics of the

language can be viewed as a mapping from a real-life domain into the belief language. This mapping provides a cognitive interpretation and face-validity to both the syntax and calculus dimensions of the language.

As the rule-based inference-engine processes rules which ultimately imply an hypothesis, a belief calculus is applied to update the posterior belief in this hypothesis. The process normally terminates when the belief in one or more hypotheses exceeds a certain pre-defined cutoff value. Ideally, we would like the system to be externally valid, namely, to assign the highest posterior belief to that hypothesis which best explains the observed fact-base. If we choose to abide to Bayesian rationality, this objective requires that the system's belief calculus be consistent with the axioms of subjective probability. However, it was shown by several authors (e.g. Heckerman, 1986) that the modular structure of the rule-based architecture is generally inconsistent with the wholistic nature of Bayesian inference. From a probabilistic standpoint, domain knowledge may be characterized by a joint distribution function F defined over the hypotheses/facts space (Pearl, 1986). Attempts to capture this knowledge through a compartmentalized rule-based architecture amount to making strong independence assumptions on F which are rarely met in practice.

The artificial intelligence literature on numeric belief update algorithms consists of two major trends. Global methods, which

are due primarily to Pearl (1986) and his colleagues at UCLA, view the knowledge-base as a space of propositional random variables, organized in a network. Pearl has shown that, for a certain class of distribution functions, there exists a method of computing posterior beliefs which is consistent with the axioms of subjective probability. Moreover, the method's run-time is polinomial in the number of nodes in the network. Similar algorithms were recently proposed by Shenoy and Shafer (1987) for the Dempster-Shafer model. Local or rule-based methods, which include MYCIN's and PROSPECTOR's belief update models, are only partially consistent with probability theory. Therefore, it is more prudent to describe a rule-based calculus as a "scoring" algorithm, a term coined originally by Cooper (1984). This algorithm accepts a set of inexact rules and a set of uncertain data, and goes on to "score" a set of competing hypotheses. There exist conditions under which the resulting scores are probabilities, but this is not always the case.

In view of the limited Bayesian rationality of rule-based inference, it is appropriate to question the merit of forwarding probabilistic research in this direction. There are several reasons, however, which make this a legitimate and potent area of inquiry. First, there exist techniques designed to transform certain wholistic evidential spaces into decomposed spaces in which rule-based belief calculi do have a Bayesian interpretation (Charniak, 1983, Schocken, 1987). Second, due to their relatively simple and "logical" structure, rule-based calculi

seem to enjoy a descriptive appeal; that is, they make sense to human experts. This is at least one reason for the fact that "at present, almost all commercially-available expert system shells are based on either EMYCIN or its fairly closed relative PROSPECTOR" (Bramer, 1986, p. 3). Indeed, following the great popularity of such shells as EMYCIN, M.1, and AL/X, rule-based belief calculi became the de-facto method of handling uncertainty in applied expert systems. Consequently, the question of how far these relatively simple and appealing methods can be pushed is interesting, both on theoretical and on practical grounds.

The present paper describes meta-enhancements to Prolog which enable it to (a) recognize and "understand" the notion of uncertain facts and inexact rules, and, (b) compute the posterior belief in hypotheses with respect to a given belief calculus. Before delving into this discussion, we wish to present a simple example which highlights the essence of what follows. The example is taken from the familiar domain of rating propspective dates listed in a "little black book." Suppose a person, denoted hereafter "dater," wishes to determine whether or not another person is a good match for a blind-date, based on a limited set of available facts. For the sake of simplicity, let's assume that the dater's knowledge-base consists of the following two rules and two facts:

```
nice_voice(X) => good_looking(X) # 0.4.                    (1)

good_looking(X) or smart(X)                                (2)
                    => date(X) # 0.8.

nice_voice(leslie) # 1.0.                                  (3)

smart(leslie) # 0.7.                                       (4)
```

This knowledge-base has the following interpretation: (1) is a wishful (and inexact) conjecture that blind-daters typically make and then learn that they should have known better. (2) is an inexact rule of thumb which models the dater's social preferences. (3) is a certain fact about Leslie. S/he sounds good over the telephone. Fact (4) is an inexact estimate of Leslie's IQ.

We see that, not unlike other domains of expertise, the dater's "knowledge" and perception of reality are heuristic and subjective, respectively. In the rule-based architecture of (1-4), this non-determinism is represented by the numbers following the # symbol. Note, however, that, barring these numbers, (1-4) may be readily translated to standard Prolog. To do this, one replaces the non-standard token => by Prolog's :- operator and reverses the direction of the two rules. These cosmetic transformations are of little theoretic interest.

Indeed, had we chosen to truncate all the degrees of belief in (1-4) starting with the # symbol, we could have asked Prolog to prove the goal:

```
date(leslie) ?
```

Prolog's response to this query will be the laconic and rather unproductive result "Yes." Under the given semantics, this means: "go ahead and date Leslie." We think that most daters would reject this black and white dichotomy in favor of a finer and more informative matcher. In particular, let's assume that (a) the # degrees of belief in (1-4) were reinstated, and, (b) a certainty-factors oriented meta-interpreter called SOLVE were available. Under these conditions, the original query may be recast as the following meta-query:

```
solve(date(leslie),Bel) ?
```

To which Prolog will answer:

```
Yes, Bel=0.56
```

Like standard Prolog, SOLVE attempts to prove the goal date(leslie), searching for facts and rules which imply this hypothesis categorically. In the process of constructing this proof, however, SOLVE also collects degrees of belief relevant to Leslie and fuses them into Bel, the posterior belief in the proposition date(leslie). In a meta-interpreter environment, the Bel variable is bound and updated on the fly, as a side-effect of the ordinary proof process.

The preceding discussion made the implicit assumption that SOLVE has a built-in belief calculus. In other words, the belief calculus is assumed to be a fixed part of SOLVE's theory. However, in view of Sterling's (1986) principles of mixing flavors, it is far more elegant and tasteful to define a stand-alone calculus, say c, and pass it on to the SOLVE meta-interpreter as a parameter. In this form, the query solve(h,Bel,c) consists of a request to confirm an hypothesis, h, and compute its posterior belief, Bel, modulo the belief calculus, c.

For example, let c1 and c2 be two complex Prolog predicates which implement the certainty-factors (CF) and the Bayesian calculus, respectively, and consider the following set of queries:

```
solve(date(leslie),c1,x1) ?
solve(date(pat),c1,y1) ?
solve(date(leslie),c2,x2) ?
solve(date(pat),c2,y2) ?
```

Suppose that the results of this experiment were x1 > y1 and x2 < y2. Let's assume further that the underlying knowledge-base as well as the corresponding CF and Bayesian degrees of belief were elicited from the same human expert. Under such circumstances, the results of the experiment clearly indicate that at least one of the belief languages under consideration failed to capture the human's preferences. This amounts to a powerful test of the empirical rationality of both languages:

excluding the change in the belief calculi and in the corresponding sets of degrees of belief, everything else is kept intact, including Leslie, Pat, the dater's preferences, and the inference-engine.

We wish to avoid here some pressing questions regarding the empirical validity of such experiments. These questions are at the heart of the broader issue of validating expert systems, a subject of a different paper. We do wish to emphasize, however, the instrumental role that meta-interpreters play in the context of such experiments. Specifically, the SOLVE meta-interpreter serves two purposes: first, it provides a clear and concise conceptualization of the experiment setup. Second, it serves as a working shell which can be instantiated with "competing" belief calculi, leading to alternative and often conflicting system recommendations. These data sets, in turn, provide important insights into the compatibility of belief calculi and their sensitivity to a variety of design changes. Research in this direction is reported in Schocken (1987).

The following section describes how rule-based knowledge is represented and parsed in the SOLVE environment. This discussion sets the stage for Section 4, where a detailed description of SOLVE's inference-engine is given.

3. Knowledge Representation

So far we have deliberately ignored the meaning of the numbers following the # symbol. The interpretation and treatment of these degrees of belief depend on our choice of a belief language. In EMYCIN and M.1, degrees of belief are elicited and represented as diagnostic certainty-factors, bearing evidence from facts to hypotheses. In Bayesian systems (e.g. PROSPECTOR), degrees of belief are causal, representing the likelihood of observing certain facts given alternative prospective hypotheses. Causal and diagnostic methods of knowledge engineering are quite different, both on cognitive and on mathematical grounds. Readers who are interested in this important "duality" are referred to Einhorn and Hogarth (1987) and to Shachter and Heckerman (1986).

This section deals with two syntactically related topics. First, we present a simple "user-oriented" language for representing inexact rules and uncertain facts. Using this language and a standard word-processor, one can create and update a knowledge-base outside the Prolog environment. Let's assume that this knowledge-base is stored in a flat file called KBASE. Next, we wish to be able to merge KBASE with SOLVE's theory through Prolog's system predicate CONSULT. This, however, requires a certain degree of parsing and pre-processing, which are also covered in this section.

Going back to the dating example, consider the following subset of an hypothetical, CF-oriented, KBASE file:

```
/* rule-base */

rich(X) => date(X) # 0.2.

age(X,Age) and Age>18 and Age<35 => date(X) # 0.3.

good_looking(X) => date(X) # 0.8.

salary(X,Salary) and Salary>75000 or
parent(X,Parent) and salary(Parent,SalaryP) and SalaryP>150000
                => rich(X) # 0.9.

/* fact base */

age(nicky,28).
parent(nicky,bob).
salary(bob,160000).
salary(nicky,20000) # 0.8.
good_looking(pat) # 0.95.
salary(pat,0).
age(pat,24).
potential_date(nicky).
potential_date(pat).
```

How can we merge this set of non-standard clauses with a standard Prolog database? ideally, we would like to simply prove the goal consult(kbase). This, however, won't work, since the KBASE syntax in incompatible with Prolog. This difference can be resolved as follows: first, enhance Prolog's syntax by adding the tokens "=>," "or," "and," and "#" to the language. Next, specify their semantics. The first modification is accomplished through the following predicate:

```
define_syntax :-  op(255,xfy,=>),
                  op(254,xfx,#),
                  op(254,xfx,or),
                  op(253,xfx,and).
```

Each application of the system predicate op(P,A,T) defines the token T as a new, non-standard Prolog operator. The precedence and associativity properties of T are given by P and A, respectively. The actual values of these arguments vary from one Prolog implementation to another and are of little interest.

Having added a bunch of non-standard clauses to Prolog, we now have to describe their intended meaning. The general strategy taken here is to convert all rules and facts into a "generic clausal form" consistent with Prolog's syntax. In particular, we wish to (a) convert inexact rules of the form <e => h # Bel> into the generic clause (h,e,Bel), and, (b) convert uncertain facts of the form <e # Bel> into the generic clause (e,true,Bel). The generic clause is important because this is the only inferential data-type that SOLVE understands. More about that, later.

Since the direction of rules and the semantics of degrees of belief vary across belief languages, each language requires a specialized parser. The remainder of this section presents a certainty-factors parser and a Bayesian parser. The section concludes with some general remarks on other functions which may be incorporated in more sophisticated parsers.

A Certainty-factors Parser: In the additive CF syntax, a diagnostic rule of the form <e => h Bel) means that e increases the belief in h by the magnitude Bel which varies from -1 to 1. If e is irrelevant to h, Bel=0. The extreme case of e being

sufficiently convincing to confirm (disconfirm) h in certainty is modeled through Bel=1 (Bel=-1). There are basically two types of certainty-factors. The CF's associated with rules (e.g. rich(x) => date(x) # 0.4) are elicited from a domain expert when the systems's rule-base is being constructed. The CF's associated with uncertain facts (e.g. salary(nicky,20000) # 0.8) are supplied through consultation.

A knowledge-base with certainty-factors is translated into generic clauses through the following parser:

```
parse(H,E,Bel) :- (E => H # Bel).                    (5)
parse(E,true,Bel) :- (E # Bel).                      (6)
parse(E,true,1) :- E.                                (7)
```

This code reads as follows: (5) matches the non-standard rule <E => H # Bel> with the clause (H,E,Bel). (6) matches the uncertain fact <E # Bel> with the clause (E,true,Bel). Finally, certain facts of the form E (with no attached degrees of belief) are defaulted by (7) to the clause (E,true,1) which reads: E is true with certainty. The latter convention allows us to freely mix certain and uncertain facts in the same knowledge-base, and, at the same time, relieves us from the tedium of assigning a 1.0 degree of belief to such certain facts as parent(nicky,bob). Instead, we let the system take care of this nuisance as a side-effect of parsing.

A Bayesian Parser: In the Bayesian language, the flow of evidence is the reverse of the CF language, reflecting causal reasoning from hypotheses to evidence. Specifically, the rule <h -> e Bel> represents the assertion that (the cause) h is a potential explanation to (the effect) e. Given this interpretation, there are many ways to define the degree of belief, Bel, measuring the "strength" of this causal implication. Indeed, the probabilistic meaning of causality has been the subject of an intense philosophical debate, and the reader is referred to Bunge (1979), Carnap (1954), and Churchman (1971) for insightful discussions of this issue.

The Bayesian calculus implemented here is based on heuristic extensions of Bayes rule. This calculus requires that each rule of the form <h => e Bel> be accompanied by three probabilities: P(h), P(e|h), and P(e|$\underline{h}$). The probability of e, P(e), need not be specified, because it is either (a) given, if e is a terminal fact, or, (b) calculated by the system through a lower-level rule of the form <e -> e' Bel'>. The most natural place to store the three probabilities associated with each rule is in the Bel parameter. Hence, we make the syntactical convention that the Bayesian degree of belief, Bel, is the three-place list [P(h),P(e|h),P(e|$\underline{h}$)]. With that in mind, the Bayesian parser is defined as follows:

```
parse(H,E,Bel) :- (H => E # Bel).                    (8)
parse(E,true,Bel) :- (E # Bel)                       (9)
parse(E,true,[0.9999,1,1]) :- E.                    (10)
```

The meaning of (8) and (9) is identical to their corresponding meaning in the CF parser, but note that the direction of the rule in the right-hand side is reversed. When the parser detects a certain fact through (10), it defaults its prior probability to 0.9999. The difference between this and the more plausible 1 is due to an uninteresting technical detail.

Similar to the CF parser, the role of (8-10) is to translate rules and facts into the generic clause (H,E,Bel) which is recognizable by the SOLVE meta-interpreter. Note that no attempt is made here to unpack the compound degree of belief into its three individual components. This task is left where it belongs -- the belief calculus level. This again illustrates how a modular design can relieve the inference-engine from unnecessary technical clutter.

Other Uses of Parsers: Thoughtful combinations of the OP and PARSE predicates can result with a great deal of design flexibility. In the present context, this flexibility allows the designer to modify the syntax of a belief language and its corresponding knowledge-bases without tinkering with the rest of the system. For example, suppose we wish to leave the CF calculus intact, and, at the same time, elicit degrees of belief that vary from -100 to 100 (this is normally done by most CF knowledge engineers). This leads to rules and facts of the form:

```
likes(X,sushi) -> date(X) # -10.
nationality(X,japan) -> likes(X,sushi) # 90.
nationality(tomo,japan).
```

Following the standard CF requirement that degrees of belief be restricted to the interval [-1,1], we can pre-process the knowledge-base as follows:

```
parse(H,E,Bel)   :-  (E -> H # Bel1),
                     Bel is Bel1/100.
parse(E,true,Bel)  :-  (E # Bel1),
                       Bel is Bel1/100.
parse(E,true,1)   :-  E.
```

One can easily envision other useful applications of PARSE beyond this trivial example. In PROSPECTOR, for example, there is a provision for representing belief in evidence through qualitative terms, e.g. "occasional," "rare," etc. Those statements are then transformed into probabilities, e.g. 0.1 and 0.01 ,respectively (Duda et al, 1977). In a similar vein, Lichtenstein and Newman (1967) concluded empirically that verbal descriptions of uncertainty may be mapped on ranges of probabilities. These verbal-numeric mappings can be made explicit as a side-effect of parsing, as follows:

```
parse(E,true,Bel) :- (E # Bel_text),
                      translate(Bel_text,Bel).

translate("occasional",0.1).
translate("rare",0.01).
etc.
```

To sum up, the parser shields the inference-engine from the syntactical idiosyncrasies of the underlying belief language. This separation enables us to elicit and represent rules and facts in a variety of forms, and, at the same time, process them through a generalized inference-engine that operates on a collection of generic clauses of the form (H,E,Bel).

4. The Inference Engine

In order to propagate degrees of belief in a network consisting of uncertain facts and inexact rules, a rule-based inference system must be capable of handling three generic types of reasoning: Boolean conditioning, sequential propagation, and parallel combination. This section describes each of these special cases of belief update schemes and provides their corresponding logic programming solutions. In the subsequent section, the three individual modules are integrated into the overall SOLVE meta-interpreter.

Let h, e_1, and e_2 be an hypothesis and two pieces of evidence with known prior belief Bel(h) and current beliefs Bel(e1) and Bel(e2), respectively. Our inference-engine must be capable of computing the posterior belief Bel(h|.) in light of any recursive combination of the following generic relationships:

Boolean conditioning: <e_1 or e_2 -> h Bel)

<e_1 and e_2 -> h Bel>

sequential propagation: $\langle e_1 \rightarrow e_2\ Bel_1 \rangle$, $\langle e_2 \rightarrow h\ Bel_2 \rangle$

Parallel combination: $\langle e_1 \rightarrow h\ Bel_1 \rangle$, $\langle e_2 \rightarrow h\ Bel_2 \rangle$

The exact specification of how to compute the posterior belief in h in any one of the above circumstances is precisely the definition of a rule-based belief calculus. Although the details of such specifications vary greatly across different calculi, the basic structure of the rule-based belief update model is quite invariant and isomorphic. This general structure is described in what follows, leaving the details for later sections.

4.1. Boolean Conditioning

Consider the categorical disjunctive rule $\langle e_1$ or $e_2 \rightarrow h \rangle$ which reads: either one of the two pieces of evidence e_1 or e_2 (known in certainty) can alone establish the hypothesis h. How does one extend this rule to situations in which either e_1 or e_2 are uncertain? this question is complicated by the observation that the uncertainty associated with these facts is not necessarily a standard probability, but, rather, an abstract measure of human belief. Kahneman and Miller (1986) have argued that, under these circumstances, the most reasonable rule for Boolean combination is the one used in the theory of fuzzy sets (Zadeh, 1965). This rule, which was implemented both in MYCIN and in PROSPECTOR, sets the belief in a disjunction (conjunction) to the maximal (minimal) belief in its constituents:

Bel(e1 or e2) = max(Bel(e1),Bel(e2))

Bel(e1 and e2) = min(Bel(e1),Bel(e2))

Once the belief in a rule's premise is established trough Boolean conditioning, the posterior belief in the rule's conclusion can be computed using sequential propagation.

4.2. Sequential Propagation

Rule-based belief calculi make the implicit assumption that the "actual" degree of belief in a rule has to change when the belief in the rule's premise changes. Specifically, let <e->h Bel(h,e)> be a rule specifying that "given e (with certainty), h is implied to a degree of belief Bel(h,e)," and let the current belief in e be Bel(e). In the process of doing rule-based inference, the premise e might be either (a) a terminal fact whose prior belief Bel(e) is specified by the user, or, (b) an intermediate "sub-hypothesis" whose current belief Bel(e|.) was already computed by the system.

Whichever category e falls in, the posterior degree of belief in the rule, denoted Bel'(h,e), is computed through a variant of the following "sequential propagation function:"

Bel'(h,e) = Fs (Bel(e),Bel(h,e))

The function Fs is monotonically increasing in both variables Bel(e) and Bel(h,e). Therefore, Fs is sometimes referred to in

the AI literature as an "attenuation function," designed to translate the uncertainty associated with a rule's premise into the uncertainty associated with the rule itself.

4.3. Parallel Combination

Let h be an hypothesis with current degree of belief Bel(h) and let $<e_1 \rightarrow h\ Bel(h,e_1)>$ and $<e_2 \rightarrow h\ Bel(h,e_2)>$ be two rules that bear evidence on h independently. The combined, posterior belief in h in light of $\{e_1,e_2\}$ is given by the following binary "parallel combination function:"

$$Bel(h|e_1,e_2) = Fp\ (Bel(h),Bel(h,e_1),Bel(h,e_2))$$

(it is implicitly assumed that $Bel(h,e_1)$ and $Bel(h,e_2)$ were already attenuated by Fs). In order to free the inference process from order and clustering effects, the function Fp is normally required to be commutative and associative. If these requirements are satisfied, the binary Fp function can be extended recursively to an n-ary parallel combination function. The details of this extension are straightforward.

The description of a belief calculus given in this section was deliberately given in skeletal terms; this abstract level of specification is all that is required by the SOLVE meta-interpreter. The actual specification of the functions <Fs,Fp,Fand,For> is made at another, meta-level of interpretation. This technique, however, requires a method for

binding a specific belief calculus <fs,fp,fand,for> to the SOLVE environment. This technique is described in the next section.

4.4. A Note on "Function Variables" in Prolog

In the preceding paragraphs, the sequential and parallel combination functions were specified using the conventional algebraic notation Y=f(X). In Prolog, this notation has no meaning. Instead, the logic programming equivalent of the computation Y=f(X) is normally the predicate f(X,Y). This goal is made to succeeds always, unifying the variable Y to the value f(X). For example, the successor function s(X)=X+1 is implemented through the predicate s(X,Y) :- Y is X+1. When we ask Prolog to prove the goal s(3,Y), Prolog succeeds and binds Y to 4 as a side-effect.

Now, things become slightly more complicated if we wish to treat the functor f itself as a variable. This is precisely what is required in the SOLVE meta-interpreter, which uses a belief calculus without knowing its exact specification. From a design standpoint, the ideal solution is to pass the four predicates <fs,fp,fand,for> as parameters to the SOLVE predicate, creating a goal of the form solve(h,Bel,fs,fp,fand,for). In this context, the predicates <fs,fp,fand,for> are meant to instantiate the variables <Fs,Fp,Fand,For> in SOLVE. However, this type of quantification is beyond the scope of first-order predicate calculus, and, consequently, is illegal in Prolog. This limitation can be overcome by second-order programming, taking

advantage of Prolog's "univ" =.. operator. Among other things, this operator may be used to bind variables to functions. For example, consider the following APPLY predicate, defined in (Sterling and Shapiro, 1986):

```
apply1(F,Xs) :- Goal=..[F|Xs], Goal.
```

The goal apply1(f,Xs) causes Prolog to apply the function f to the argument list Xs. For example, the goal apply1(s,[3,Y]) will succeed, resulting with Y=4.

In this paper we define a more powerful version of APPLY, as follows:

```
apply(Predicate,Args) :- Predicate=..PredList,
                         append(PredList,Args,GoalList),
                         goal=..GoalList,
                         call(Goal).
```

Defined that way, the first argument of apply, Predicate, can be either an atomic symbol naming a predicate, or, alternatively, a term representing a predicate with some of its arguments supplied. For example, apply(s,[3,Y]) will yield Y=4, and so will apply(s(3),[Y]). As yet another example of the utility of APPLY, consider the following numeric computation of the square-root function, using Newton's approximation formula:

```
sqrt(X,Y) :- apply(newton(0.01),[X,Y]).
newton(Epsilon,X,Y) :- iterate(Epsilon,X,Y,1).
iterate(Epsilon,X,Y,Y) :- Diff is X-Y*Y,abs(Diff,Z),Z=<Epsilon,!.
iterate(Epsilon,X,Y,Z) :- NewZ is (X/Z+Z)/2,
                          iterate(Epsilon,X,Y,Newz)
```

Defined that way, the parameter of the NEWTON predicate, currently set to 0.01, specifies the precision level of the SQRT function. That is, Y is guranteed to be within a 0.01 neighborhood of the true value of $\sqrt{X}$. In this example, sqrt(4,Y) will yield Y=2.0006.

To sum up, we see that the term representing the predicate in our definition of APPLY is the equivalent of a closure in a Lisp-based functional language.

The programming techniques discussed in this section have general implications to software engineering beyond the context of this paper. We have chosen to present this material here because second-order programming is used extensively in the SOLVE architecture. In particular, the APPLY predicate plays a central role in mixing flavors, i.e. adding functionalities to the "vannila" meta-interpreter. This analogy becomes clearer in the next section.

4.5. Putting it All together

The theory of SOLVE consists of a recursive union of all the modules described thus far, namely parsing, Boolean conditioning, sequential propagation, and parallel combination. The complete definition of SOLVE is as follows:

```
solve(true,1,Fs,Fp,Fand,For) :- !.                                    (11)

solve((H1 or H2),Bel,Fs,Fp,Fand,For) :-                               (12)
                          solve(H1,Bel1,Fs,Fp,Fand,For),
                          solve(H2,Bel2,Fs,Fp,Fand,For),
                          apply(For(Bel1,Bel2),[Bel]).

solve((H1 and H2),Bel,Fs,Fp,Fand,For) :-                              (13)
                          solve(H1,Bel1,Fs,Fp,Fand,For),
                          solve(H2,Bel2,Fs,Fp,Fand,For),
                          apply(Fand(Bel1,Bel2),[Bel]).

solve(H,Bel,Fs,Fp,Fand,For) :-
                  parse(H,_Belp),!,                                   (14)
                  bagof(Belx,                                         (15)
                        (parse(H,E,Bel_rule),                         (16)
                         solve(E,Bel_e,Fs,Fp,Fand,For),               (17)
                         apply(Fs,[Bel_e,Bel_rule,Belx)),             (18)
                       Bels),                                         (19)
                    apply(Fp,[Belp,Bels,Bel]).                        (20)

solve(E,1,Fs,Fp,Fand,For) :- E,!.

solve(E,0,Fs,Fp,Fand,For).
```

The base-fact (11) of SOLVE, which is ground, assigns a belief of 1 to the constant hypothesis "true." The subsequent handling of Boolean conditioning in (12-13) is self-explanatory. In (14), PARSE is used to check if the hypothesis H is present in the knowledge-base, and, if so, to bind Belp to its prior degree of belief. The BAGOF predicate accomplishes a few things. First, it looks (through parsing) for all the rules <E => H Bel_rule> whose conclusion is H (16). For each such rule, SOLVE is applied

recursively to compute the posterior belief in the premise, E, yielding Bel_e (17). This "current belief," in turn, is used by Fp to attenuate the original degree of belief, Bel_rule, into Belx (18). Attenuated degrees of belief are strung together (via BAGOF) into the list Bels (19).

The "punch line" of SOLVE is (20). When we get to this point, the list Bels consists of all the attenuated degrees of belief associated with all the rules whose conclusion is H. Since this list is constructed recursively, Bels incapsulates all the evidence that SOLVE drew from all the reasoning chains whose ultimate conclusion is H. At that point, the parallel combination function Fp is applied to fuse this information with the prior belief Belp, yielding the ultimate outcome of SOLVE, i.e. the posterior belief, Bel.

To sum up, SOLVE(H,Bel,Fs,Fp,Fand,For) implements an exhaustive depth-first search, pruning all the rules and facts which bear evidence on H, either directly or indirectly. As a side-effect of this process, the program computes the posterior belief in H modulo the belief calculus <Fs,Fp,Fand,For>. When SOLVE branches horizontally, Fp is used to combine the degrees of belief originating from rules whose direct conclusion is H. When SOLVE backtracks from a vertical recursive call, Fs is used to synthesize the belief committed to H from lower-levels of reasoning. If a Boolean "fork" is encountered, either Fand or

For are applied to compute the posterior belief coming out of the fork.

The reader has probably noticed that the predicates <Fs,Fp,Fand,For> are still unspecified. This was done in purpose, in order to highlight the modularity and top-down design of SOLVE. Indeed, one motivation for writing this paper was to demonstrate the ease by which a belief calculus can be added to or modified within the SOLVE environment. This is accomplished in a completely orthogonal manner, i.e. without tinkering with any other part of the meta-interpreter. To illustrate this point, we now proceed to define two examples of well-known rule-based belief calculi. In the modular SOLVE environment, this amounts to no more than specifying the theory of the predicates <Fs,Fp,Fand,For>.

5. Rule-Based Belief Calculi

This section gives Prolog implementations of the CF calculus and an ad-hoc Bayesian calculus. These models are presented verbatim, and no attempt is made here to either defend their cognitive appeal or argue for or against their normative justification. The literature is by now rife with probabilistic analyses and commentary of this sort, e.g. Heckerman (1986), Grosof (1986), and Schocken and Kleindorfer (1987).

5.1. The Certainty-factors Calculus

Following its great popularity in applied expert systems, the certainty-factors calculus has evolved into several forms, all of which might be easily incorporated into the SOLVE architecture. The calculus discussed here adheres to the original model, described in detail by Buchanan and Shortliffe (1984).

Sequential combination: The CF associated with the diagnostic rule <e => h CF(h|e)> is elicited from a domain expert under the assumption that the premise e is known with certainty. When the belief in e is less than certainty, the CF calculus attenuates the rule's degree of belief through the following sequential propagation function:

$$CF'(h|e) = \begin{cases} CF(h|e) * CF(e) & \text{If } CF(e) > 0 \\ 0 & \text{otherwise} \end{cases}$$

This function is implemented in Prolog as follows:

```
cf_s(Bel_e,Bel_rule,Bel) :- max(0,Bel_e,Bel_max),
                            Bel is Bel_rule * Bel_max.
```

Parallel combination: When two rules <e_1->h CF(h|e_1)> and <e_2->h CF(h|e_2)> bear evidence on h independently, their compound increased belief in h in light of {e_1,e_2} is computed through the binary CF parallel combination function:

$$CF(h|e_1,e_2) = \begin{cases} \text{if both CF's are positive:} \\ CF(h|e_1) + CF(h|e_2)*(1-CF(h|e_1)) \\ \\ \text{if both CF's are negative:} \\ -(|CF(h|e_1)|+|CF(h|e_2)|*(1-|CF(h|e_1)|) \\ \\ \text{if } CF(h|e_1)>0 \text{ and } CF(h|e_2)<0 \\ \dfrac{CF(h|e_1) - CF(h|e_2)}{1-\min(|CF(h|e_1)|,|CF(h|e_2)|)} \end{cases} \quad (21)$$

The Prolog implementation of this function is as follows:

```
cf_p_2(X,Y,Z) :- X>=0, Y>=0,
                 Z is X+Y*(1-X),!.

cf_p_2(X,Y,Z) :- ((X=<0,Y>=0) ; (X>=0,Y=<0)),
                 abs(X,A), abs(Y,B), min(A,B,C),
                 Z is (X+Y)/(1-C),!.

cf_p_2(X,Y,Z) :- X<0, Y<0,
                 abs(X,A), abs(Y,B),
                 Z is -(A+B*(1-A)),!.
```

An inspection of (21) reveals that cf_p_2 is both commutative and associative. Hence, (21) might be applied recursively to compute the compound evidential impact of any finite set of independent rules. The resulting n-ary expansion of cf_p_2 is as follows:

```
cf_p(_,[],0).
cf_p(_,[X|Xs],Bel) :- cf_p(_,Xs,Bel_Xs),                    (22)
                      cf_p_2(X,Bel_Xs,Bel).
```

The explicit omission of the first variable in (22) underscores the fact that the CF language ignores prior beliefs. This can be seen clearly in the definition of the base-fact of cf_p, which models the "state of insufficient reason" (Savage, 1954). This case, which is characterized by an empty set of diagnostic facts, causes cf_p to assign a posterior belief of 0 to the hypothesis in question. This is consistent with the additive CF rationale, in which the absence of any relevant evidence on h causes the belief in h to neither increase nor decrease. In a Bayesian language, one would normally model this case by setting the posterior belief in the hypothesis to its prior belief.

Boolean Conditioning: the CF definitions of the functions Fand and For are as follows:

```
cf_and(Bel1,Bel2,Bel) :- min(Bel1,Bel2,Bel).
cf_or(Bel1,Bel2,Bel) :- max(Bel1,Bel2,Bel).
```

5.2. An Ad-Hoc Bayesian Calculus

The ad-hoc Bayesian (AHB) calculus described below operates on causal rules of the form <h -> e Bel>. Recalling section 3, the degree of belief Bel associated with the rule h->e is assumed to be a three-place list Bel=[x1,x2,x3] with x1=P(h), x2=P(e|h), and x3=P(e|$\underline{h}$).

Sequential propagation: The literature contains several heuristic procedures for sequential belief update, e.g. Jeffries rule of conditioning (Shafer, 1981) and PROSPECTOR's interpolation function (Duda et al, 1977). For the sake of brevity, we choose to describe here a simple interpolation function, discussed by Wise (1986). This function defines the "attenuated" degree of belief P'(e|h) as a weighted average of P(e|h) and P(ē|h), weighted by P(e) and P(ē), respectively:

$$P'(e|h) = P(e|h) * P(e) + (1-P(e|h)) * (1-P(e)) \qquad (23)$$

The AHB sequential propagation function is implemented as a 3-place Prolog predicate called ahb_s. The first two "input" variables of ahb_s are the rule's degree of belief [P(h),P(e|h),P(e|h̄)] and the belief in the rule's premise P(e). The third "output" variable is the attenuated, 3-place degree of belief, [P(h),P'(e|h),P'(e|h)], which is computed through (23). Note that ahb_s leaves the prior P(h) intact. The variables naming in ahb_s is as follows: the list [P(h),P(e|h),P(e|h̄)] and the scalars P(h), P(e), P'(e|h), and P(e|h̄) are denoted by [P0,Q1,Q2], P0, Bel_e, P1, and P2, respectively.

```
ahb_s([P0,Q1,Q2],Bel_e,[P0,P1,P2]) :-
                              P1 is Q1*Bel_e + (1-Q1)*(1-Bel_e),
                              P2 is Q2*Bel_e + (1-Q2)*(1-Bel_e).
```

Parallel combination: Let <h -> e_1 Bel_1> ,..., <h -> e_n Bel_n> be n causal rules with Bel_i=[P(h),P(e_i|h),P(e_i|$\underline{h}$)]. The posterior belief in h in light of the evidence {e_1,...,e_n} is computed through the following version of (the commutative and associative) Bayes rule:

$$\text{product\_odds} = \frac{P(e_1|h)}{P(e_1|\underline{h})} * \ldots * \frac{P(e_n|h)}{P(e_n|\underline{h})}$$

$$\text{odds} = \text{product\_odds} * \frac{P(h)}{P(\underline{h})} \qquad (24)$$

$$P(h|e_1,\ldots,e_n) = \text{odds} / (1+\text{odds})$$

Let the the set {Bel_1,...,Bel_n} and the scalars P(h) and P(h|e_1,...,e_n) be the list Bels and the atoms Prior and P, respectively. Given this naming convention, the Prolog implementation of (24) is the predicate ahb_p, defined as follows:

```
ahb_p([Prior|_],Bels,P) :-
                    mult(Bels,Product_Odds),
                    Odds is (Prior/(1-Prior)) * Product_Odds,
                    P is Odds/(1+Odds).

mult([],1).
mult([[X1,X2]|Xs],Product) :- mult(Xs,Bel_Xs),
                              Product is (X1/X2)*Bel_Xs.
```

A more efficient, tail-recursive version of MULT can be defined as follows:

```
mult(List,Product) :- multl(List,1,Product).
multl([],Accumulator,Accumulator).
multl([[_,X1,X2]|Xs],Acumulator,Product) :-
                          NewAccumulator is Accumulator*X1/X2,
                          multl(Xs,NewAccumulator,Product).
```

Boolean Conditioning: the definitions of the functions Fand and For in the ad-hoc Bayesian model are equivalent to their CF versions:

```
ahb_and(Bel1,Bel2,Bel) :- min(Bel1,Bel2,Bel).
ahb_or(Bel1,Bel2,Bel) :- max(Bel1,Bel2,Bel).
```

6. Cooking Instructions

Wirth's (1976) design principle of <programs = algorithms + data structures> is well known. In the context of expert systems, this translates into <expert system = inference mechanism + knowledge-base>. In this paper we have taken the modularity principle one step further, achieving what may be described symbolically as <inference mechanism = inference engine + belief calculus>. The resulting SOLVE environment is basically a collection of modules that can be intermixed without having to tinker with the theory of any one individual module.

The practice of incremental enhancements of meta-interpreters was analyzed by Sterling (1986). This analysis, which draws its

terminology from object programming, suggests that Prolog meta-interpreters are analogous to Lisp Flavors. Using Sterling's language (which is underlined), the modules PARSE and <Fs,Fp,Fand,For> are orthogonal enhancements to the SOLVE flavor, in that the computations necessary for incorporating them are completely separate. The PARSE predicate amounts to a behavioral enhancement: it extends the computation performed by SOLVE without changing the meta-goal of the enhanced meta-interpreter. This is done simply by adding the parsing predicates to SOLVE's theory. The flavor SOLVE(H,Bel,Fs,Fp,Fand,For) is a structural enhancement of the vannila flavor SOLVE(H), in that the extra arguments <Fs,Fp,Fand,For> (which are "initialized" to a specific belief calculus <fs,fp,fand,for>) are used to compute Bel as H is being solved.

So, now that all the ingredients have been provided, the creation of a rule-based inference system is merely a matter of mixing flavors. Let p and q be two Prolog predicates whose extended theory is stored in two files named "p" and "q" (the extended theory of p includes p's theory and the theory of all the predicates mentioned in p's theory). In what follows, when we say "add p to q" or "mix p and q" we mean "prove the goals consult(p) and consult(q)."

With that in mind, to prepare a CF-oriented inference system, follow this set of instructions:

1. Create a CF-oriented knowledge-base and save it in a file called KBASE
2. Prove the goal define_syntax
3. Mix the cf parse predicate with the SOLVE flavor
4. Add the predicates cf_s, cf_p, cf_and, cf_or
5. Mix the resulting inference system with the knowledge-base KBASE
6. Confirm the hypothesis h and compute its posterior certainty-factor by proving the meta-theorem solve(h,Bel,cf_s,cf_p,cf_and,cf_or).

To prepare a Bayesian-oriented inference system, follow this set of instructions:

1. Create a Bayesian-oriented knowledge-base and save it in a file called KBASE
2. Prove the goal define_syntax
3. Mix the ahb parse predicate with the SOLVE flavor
4. Add the predicates ahb_s, ahb_p, ahb_and, ahb_or
5. Mix the resulting inference system with the knowledge-base KBASE
6. Confirm the hypothesis h and compute its posterior (ad-hoc) Bayesian belief by proving the meta-theorem solve(h,Bel,ahb_s,ahb_p,ahb_and,ahb_or).

As far as Prolog is concerned, the fully instantiated SOLVE meta-interpreter is yet another predicate. Therefore, one can blend SOLVE with standard Prolog in order to implement the typical bits and pieces which make up full-blown expert systems. For example, let's go back to our dubious dating system. Perhaps the most useful output of this application would be a reorganized version of the little black book, sorted in decreasing order of composite

attractiveness. This book may be prepared by the following (CF-oriented) predicate:

```
create(Book) :- bagof((X,Rating),
                  (potential_date(X),
                   solve(date(X),Rating,cf_s,cf_p,cf_and,cf_or)),
                   Xs),
                sort(Xs,Book).
```

Given the database described in section 3 (consisting of only two potential dates), the goal create(Book) will yield the response:

```
Book = [(pat,0.832),(nicky,0.426)].
```

The casual nature of the dating example should not be confused with the underlying seriousness of the SOLVE meta-interpreter. Consider, for example, a medical diagnosis application. In this context, potential dates and their perceived characteristics correspond to prospective diseases and symptom manifestations, respectively. Sub-hypotheses, like "rich(X)," correspond to clinical syndromes or intermediate diagnoses. Finally, dating rules are analogous to text-book medical knowledge and heuristic inferences of experienced experts. Under this interpretation, the evaluation of a propspective date is analogous to the diagnosis of a certain patient. In this context, the goal create(Book) should probably be renamed to rank(Diseases). Given a certain knowledge-base and a set of symptoms (stored in KBASE), this goal gives a list of all the potential diseases that this patient might have, in decreasing order of likelihood.

7. Conclusions and Future Research

The validity of alternative belief languages can be investigated in two different and complementary methodologies. The analytic approach is chiefly concerned with comparing belief calculi to well-known normative criteria, e.g. probability theory or predicate logic. This line of research leads quite clearly to the realization that, not unlike the humans that they attempt to model, all rule-based belief calculi contain varying degrees of normative violations. Nonetheless, the extent of these violations is not well understood, and the sensitivity of the system's advice to such violations is still an open question.

In spite of their normative deficiencies, rule-based belief calculi are widely-used in commercial expert systems. Moreover, it might be that a careful design of the underlying knowledge-base might ensure that normative violations are kept to a minimum. With that in mind, there is a crucial need for an empirical methodology for investigating the external validity of alternative belief calculi. This line of research will simulate experimental settings in which the expertise of human subjects is elicited and represented via different belief languages. The experiments will then pit the systems' recommendations with (a) the judgment of the humans that they claim to model, and, (b) an external norm, such as the "true state of the world."

There exist very few empirical studies related to expert systems, and the comparative study of alternative belief languages is no exception. Most of the work in this area was carried out during the last two years, e.g. Mitchell (1986), Yadrick et al (1986), Wise (1986), and Schocken (1987). These studies attempt to understand the conditions under which one belief language performs better than another. Therefore, they have important prescriptive implications on knowledge engineering.

One limitation that inhibited more research in this direction has been a lack of a common benchmark environment. Such environment ought to simulate rule-based inference on the one hand, and, on the other hand, allow a great deal of design flexibility in terms of experimenting with alternative belief calculi. We feel that the SOLVE meta-interpreter presented in this paper is a first step toward closing this gap. We hope that other people will modify this meta-interpreter to meet their own research needs, and that this will promote further understanding of the empirical validity of expert systems.

Appendix

```
/*  The following is a listing of all the miscellaneous
predicates referred to in the paper  */

apply(Predicate,Args) :- Predicate=..PredList,
                         append(PredList,Args,GoalList),
                         goal=..GoalList,
                         call(Goal).

bagof(X,G,_) :- asserta(found(mark)),G,asserta(found(X)),fail.
bagof(_,_,L) :- collectFound([],M),!,L=M.
collectFound(Lin,Lout):- getNext(X),!,collectFound([X|Lin],Lout).
collectFound(Lin,Lin).
getNext(X) :- retract(found(X)),!,not(X==mark).

max(X,Y,X) :- X>=Y.
max(X,Y,Y) :- X<Y.

min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.

abs(X,Z) :- X<0, Z is -(X).
abs(X,X) :- X>=0.

append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).

/* the following predicate sorts a list of pairs [X,Y] in
decreasing order of Y */

sort([X|Xs],Ys) :- sort(Xs,Zs), insert(X,Zs,Ys).
sort([],[]).
insert([X1,X2],[],[[X1,X2]]).
insert([X1,X2],[[Y1,Y2]|Ys],[[Y1,Y2]|Zs]) :-
             X2>Y2, insert([X1,X2],Ys,Zs).
insert([X1,X2],[[Y1,Y2]|Ys],[[X1,X2],[Y1,Y2]|Zs]) :- X2=<Y2.
```

References

Alvey, P. L., Myers, C. D., and Greaves, M. F., "An Analysis of the Problems of Augmenting a Small Expert System," in Bramer, M. A. (ed.) Research and Development in Expert Systems, Cambridge University Press, 1986, pp. 61-72.

Baldwin, J. F., and Monk, M. R. M., "SLOP - a System for Support Logic Programming," I.T.R.C. research report, University of Bristol, 1986.

Bramer, M. A., "Expert Systems: the Vision and the Reality," in Bramer, M. A. (ed.), Resaerch and Development in Expert Systems, Cambridge University Press, 1986, pp. 1-12.

Bunge, M., "Causality and Modern Science," New York: Dover Publications, 1979.

Carnap, R., Logical Foundations of Probability, Chicago: University of Chicago Press, 1954.

Charniak, E., "The Bayesian Basis of Common Sense Medical Diagnosis," Proceedings of the National Conference in Artificial Intelligence, 1983, 3, pp. 70-73.

Churchman, C. W., "The Design of Inquiring Systems," Basic Books, 1971.

Clark, K. L. and McCabe, F. G., "Prolog: a Language for Implementing Expert Systems," in Hayes, Michie, D. and Pao, W. H. (eds.), Machine Intelligence 10, Ellis Horwood, 1982, pp. 455-470.

Cohen, P., Davis, A., at al, "Representativeness and Uncertainty in Classification Systems," The AI Magazine, Fall 1985, pp. 139-149.

Cooper, G. F., "NESTOR: a Computer Based Medical Diagnostic Aid that Integrates Causal and Probabilistic Knowledge," unpublished Ph.D. Dissertation, Stanford University, 1984.

Davis R., and King, J. J., "The Origin of Rule-Based Systems in AI," in Buchanan, B. G., and Shortliffe, E. H. (Eds.), Rule-Based Expert Systems, Addison-Wesley, 1984, pp. 20-52.

Dincbas, M., Metacontrol of Logic Programs in METALOG, Proc. of FGCS, Tokyo, Japan, November, 1984, pp. 361-370.

Duda, R. O., Hart, P. E., and Nilsson, N. J., "Development of a Computer-Based Consultant for Mineral Exploration," SRI International Projects 5821 and 6415, october 1977.

Einhorn, H. J. and Hogarth, R. M., "Decision Making: going Forward in Reverse," Harvard Business Review, January-February 1987, pp. 66-70.

Grosof, B. N., "Evidential Information as Transformed Probability," in Lemmer, J. F., and Kanal, L. (eds.), Uncertainty in Artificial Intelligence. North Holland, 1986.

Hammond, P., "Micro-Prolog for Expert Systems, Chapter 11 in Clark, K. L. and McCabe, F. G. (eds.), "Micro-Prolog: Programming in Logic," Prentice-Hall, 1984.

Heckerman, D. E., "Probabilistic Interpretation for MYCIN's Certainty Factors," in Lemmer, J. F., and Kanal, L. (eds.), Uncertainty in Artificial Intelligence. North Holland, 1986.

Heckerman, D. E., "The Myth of Modularity in Rule-Based Systems," in: Proceedings of the 2nd Special Conference on Uncertainty in Artificial Intelligence, AAAI Conference, Philadelphia, PA, 1986, pp. 115-122.

Kahneman, D. and Miller, D. T., "Norm Theory: Comparing Reality to its Alternatives," Psychological Review, Vol. 93,No. 2, 1986, pp. 136-153.

Lichtenstein, S., and Newman, J. R., "Empirical Scaling of Common Verbal Phrases Associated with Numerical Probabilities," Psychonomic Science, 9, 1967, pp. 563-564.

Mitchell, D. H., "The Shape Experiment," Manuscript, Psychology Department, Northwestern University, 1986.

Newell, A., "Production Systems: Models of Control Structure," in Chase, W., (ed.), Visual Information Processing, New York: Academic Press, 1973, pp. 463-526.

Pearl, J., "Fusion, Propagation, and Structuring in Belief Networks," Artificial Intelligence, September, 1986.

Pereira, L., "Logic Control with logic," Proc. of the First International Logic Programming Conference, Marseille, 1982, pp. 9-18.

Savage, L. J., The Foundations of Statistics, Wiley, 1954.

Shachter, R. D. and Heckerman, D. E., "A Backwards View for Assessment," in Lemmer, J. F., and Kanal, L. (eds.), Uncertainty in Artificial Intelligence. North Holland, 1986.

Schocken, S., "On the Rational Scope of Probabilistic Rule-Based Inference Systems," in: Proceedings of the 2nd Special Conference on Uncertainty in Artificial Intelligence, AAAI Conference, Philadelphia, PA, 1986.

Schocken, S., "On the Underlying Rationality of Non-Deterministic Rule-Based Inference Systems: a Decision Sciences Perspective," Ph.D. Dissertation, Decision Sciences Working Paper 87-04-02, The Wharton School of the University of Pennsylvania, 1987.

Schocken, S. and Kleindorfer, P. R., "Artificial Intelligence Dialects of the Bayesian Belief Language," Center for Research on Information Systems (CRIS) Working Paper #160, New York University, 1987.

Shafer, G., A Mathematical Theory of Evidence, Princeton University Press, 1976.

Shafer, G., "Jeffrey's Rule of Conditioning," Philosophy of Science, September, 1981, pp. 337-362.

Shafer, G. and Tversky, A., "Languages and Designs for Probability Judgment," Cognitive Science 9 (1985), pp. 309-339.

Shenoy, P., and Shafer, G., "Propagating Belief Functions with Local Computations," IEEE Expert 1, 1987, pp. 43-52.

Shapiro, E. Y., "Logic Programs With Uncertainties: a Tool for Implementing Rule-Based Systems," IJCAI, 1983, Krlsruhe, West Germany, pp.529-532.

Shortliffe, E. H., and Buchanan, B. G., "A Model of Inexact Reasoning in Medicine," in: Buchanan, B. G., and Shortliffe, E. H. (Eds.), Rule-Based Expert Systems, Addison-Wesley, 1984.

Sterling, L. S., "Meta-Interpreters: the Flavors of Logic Programming?" Proc. of the Workshop on the Foundations of Deduction, Databases, and Logic Programming, Washington, D.C., August, 1986.

Sterling, L. S. and Lalee, M., "An Explanation Shell for Expert Systems," Computational Intelligence, 1986 (to appear).

Sterling, L. S. and Shapiro, E. Y., "The Art of Prolog," The MIT Press, 1986, pp. 280-283.

Wise, B. P., "Experimentally Comparing Uncertain Inference Systems in Probability," in: Proceedings of the 2nd Special Conference on Uncertainty in Artificial Intelligence, AAAI Conference, Philadelphia, PA, 1986b, pp. 319-332.

Wirth, N., "Algorithms + Data Structures = Programs," Englewood Cliffs, N.J.: Prentice-Hall, 1976.

Yadrick, R. M., Perrin, B. M., Vaughan, D. S., Holden, P. D., and Kempf, K. G., "Evaluation of Uncertain Inference Models I: Prospector," in: Proceedings of the 2nd Special Conference on Uncertainty in Artificial Intelligence, AAAI Conference, Philadelphia, PA, 1986b, pp. 333-337.

Zadeh, L. A., "Fuzzy Sets," Information and Control, 8, 1965, pp. 338-353.