# 8

# YAPES: Yet Another PROLOG Expert System

T. B. Niblett
The Turing Institute and University of Strathclyde,
Glasgow, UK

## 1. INTRODUCTION

YAPES is an expert system shell developed at the Turing Institute. It provides inference and explanation facilities, and incorporates a novel form of plausible inference.

YAPES is a specialized *interpreter* for logic programs. Figure 1 illustrates its top level structure. A PROLOG interpreter (or compiler) executes such programs consisting of sets of *Horn clauses*, a form of first-order logic. The YAPES system also executes such programs, as well as programs in an extended version of Horn clause logic which uses *certainties* as truth values, rather than just true and false.

The construction of the YAPES interpreter allows it to provide explanations of its reasoning, as well as asking questions from users and other knowledge sources. The explanations provided include *why explanations* in the sense of 'why are you asking this question', *how explanations* in the sense of 'how did you reach this conclusion' and *why not explanations* in the sense of 'why did this goal fail'.

The YAPES interpreter is flexible and can call the PROLOG interpreter to execute goals directly. Similarly the PROLOG interpreter can call the YAPES interpreter. This allows a mixture of *object* and *meta-level* inference, and provides a smooth integration with the PROLOG environment.

In Section 2 the rationale behind YAPES is described in more detail and the advantage of using logic as a representation language discussed. Section 3 describes the facilities provided by YAPES, and contains some discussion of the theoretical issues involved. Section 4 presents an annotated transcript of a YAPES session using a simple knowledge base for travel expense claims. Finally in Section 5 we discuss the advantages of the approach taken.

Many of the ideas incorporated in the program are not new and derive much from the work of Kowalski (1979), Shapiro (1983b), Sergot (1983) and others. In particular many of the utilities incorporated in the

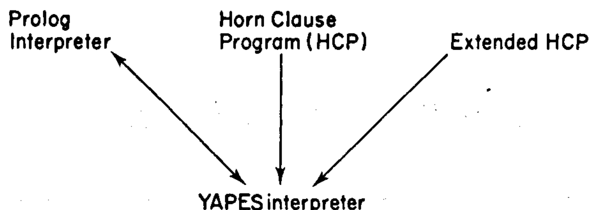


Figure 1. YAPES as an interpreter for logic programs.

program were written by Richard O'Keefe and Lawrence Byrd and I am greatly indebted to them.

## 2. THE RATIONALE BEHIND YAPES

Our intention has been to provide an expert system 'shell' containing all the normal trace and explanation facilities, and with the provision of a sophisticated plausible reasoning facility.

The best way of looking at YAPES is as a new interpreter for rather special PROLOG programs which provides the kind of user interaction needed in expert systems work.

The program is implemented in PROLOG and a major concern has been to keep the form of the knowledge-base as close to the underlying language (Horn clause logic) as possible. In fact the only extension to pure Horn clause logic is the use of a plausible reasoning mechanism, which preserves the clear semantics of the logic and is *upwards compatible* in the sense of being a conservative extension. There are several compelling reasons for taking this approach:

1. The semantics of logic programs are well understood. This permits the development of elegant and general algorithms for debugging and modifying programs which can be directly applied to expert system knoweldge-bases (e.g. Shapiro, 1983b).

2. Most commercial expert systems shells available at present lack flexibility. It is often difficult to modify the search strategy used by the shell, or to perform computations not explicitly catered for by the designers. Logic programming is far more general and by keeping our shell as close as possible to logic programs this generality can be incorporated while still allowing the system debugging and tracing facilities to be used.

3. By using the same language for the knowledge-base and the underlying program we can take advantage of the reflection principle espoused by Weyhrauch (1980) and merge the *object* and *meta-level* languages where appropriate.

### 2.1. Logic programs as a basis for expert systems

The basis of our approach is that a logic program, and the results of its execution in the form of a proof tree (see below) can be used as a knowledge-base and can provide humanly understandable explanations of how the knowledge is used by the system. Furthermore the simplicity and generality of the 'data-structures' involved allow an inexperienced user to learn the use of the system fairly quickly.

We are *not* arguing that the PROLOG language is easy to learn in its full generality. From our experience this is not the case. For the majority of *expert system* applications however, where the manipulation and creation of complex data structures is not required, the language is very straightforward.

### 2.2. The form of a logic program

We shall discuss fairly briefly the form of logic programs; this material is useful for a technical appreciation of the plausible reasoning mechanism discussed later, but not essential for the reader who wishes merely to discover what YAPES *does*.

A logic program consists of a (non-empty) set of *Horn clauses*, together with a *goal statement*. A Horn clause has the form $A \leftarrow B_1, \ldots, B_n$ where $A$ and $B_i$ are *literals*. A literal is of the form $p(T_1, \ldots, T_m)$ $(m \geq 0)$where $p$ is a *predicate name* and the $T_i$ are *terms*. A term is a *variable*, an *atom*, a *number* or an expression of the form $f(T_1, \ldots, T_m)$ $(m > 0)$ where $f$ is a *function symbol* and the $T_i$ are terms. We shall follow the PROLOG convention throughout that variables are indicated by an initial capital letter, and predicate names, function symbols and atoms by an initial lower-case letter. The goal statement of a program is written $\leftarrow G$ where $G$ is a literal.*

A Horn clause $A \leftarrow B_1, \ldots, B_n$ is interpreted in first-order logic as the assertion

$$\forall_{x_1}, \ldots, x_n\{B_1 \,\&\, \cdots \,\&\, B_n \rightarrow A\}$$

where $\rightarrow$ is material implication and $x_1, \ldots, x_n$ are variables occurring in the literals of the clause.

### 2.3. Unification

The central operation in logic programming is *unification*. Unification occurs between two literals, and involves finding a (possibly empty) substitution for the variables in the literals which makes them identical. For example the literal $f(g(X), h(h(Y)))$ unifies with the literal $f(U, h(V))$ with the substitution set $\{g(X)/U, h(Y)/V\}$. It has the useful

* A conjunction of literals $\leftarrow G_1, \ldots, G_n$ is often used in PROLOG. This is equivalent to having a single goal $\leftarrow G$ and adding the pseudo-goal $G \leftarrow G_1, \ldots, G_n$.

property in practice of allowing us to abstract the *control* of a logic program from the *flow of data* as we shall see later.

### 2.4. Execution of a program

A logic program executes successfully if the initial goal statement $\leftarrow G$ can be reduced to the empty goal ($\leftarrow$). This reduction is accomplished in stages as follows; at each stage we have a goal statement $\leftarrow G_1, \ldots, G_n$, one such goal statement $G_i$ is chosen* and *unified* with the head of one of the clauses in the program. If for the chosen goal $G_i$ there is more than one clause that can be unified, we have a *choice point.* If no $G_i$ can be unified with any clause then the execution must backtrack to a previous choice point, where an alternative clause can be found to match. This unification produces a substitution for the variables in the goal statement and in the body of the clause concerned. The goals in the body of the clause are then substituted in the goal statement in place of the goal $G_i$. Execution terminates successfully when there are no goals left.

Figure 2a demonstrates the execution of a simple program without variables. Figure 2b illustrates how this execution can be displayed as a

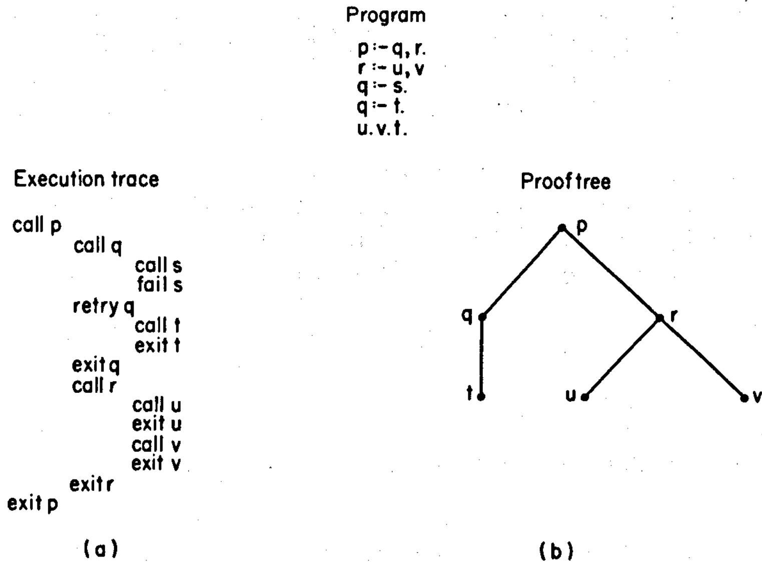


Figure 2

tree. We shall call such trees *proof trees.* We can see that information about the order in which unifications took place is omitted from such trees. Also any information about unifications which were tried and later had to be undone and another choice of clause tried has been omitted.

* In PROLOG the goal chosen is always the leftmost goal $G_1$.

This is one very useful property of logic programs. The output of a successful execution has only the relevant information about a proof: all information about how the proof tree was constructed is omitted. This ability to extract only relevant information is of great utility when it comes to providing explanations in an expert system context.

In the next section we shall see how the particular properties of the PROLOG interpreter also allow us to provide information to the user during execution.

**2.5. Reasoning about program behaviour**

As we have seen, the result of a program's execution is a proof tree. The PROLOG language allows us to create and manipulate such trees by virtue of its *meta-logical* features.

These features allow us to regard clauses and literals as terms, and to determine the current status of PROLOG objects to see whether variables are instantiated or not. In addition we can modify a program during its execution by creating new clauses (and terms, etc.).

This ability of PROLOG to create an internal representation of itself is critical in creating an expert system. Of course it is possible to do this in any language. One can for example create a PASCAL interpreter in PASCAL and simulate the execution of PASCAL programs with it. There are three important differences however.

1. The representation in PROLOG is much shorter, and therefore much easier to work with. It is possible to write a useful PROLOG interpreter in PROLOG in less than one page of code. To do this in PASCAL would be a major achievement.

2. The computation in PROLOG is easily representable as we have seen. It would not be possible to do the same in a procedural language which relies largely on side-effects.

3. The use of unification allows us to write interpreters for logic programs in PROLOG that reveal the important information, about control flow and how data-structures are to be created, without having to describe in detail the way data will be manipulated.

The program in Figure 3 illustrates an interpreter in PROLOG for PROLOG which maintains a stack of goals indicating the branch of the proof tree that the interpreter is currently investigating. This allows a user to question the system as to why it is pursuing a particular goal. This program is a complete interpreter for pure PROLOG (without the cut). Built-in system predicates are directly executed via the *call* predicate. The PROLOG system knows which goals are system goals via the *system* predicate. The meta-level predicate clause returns a clause in the program. Its two arguments are the head and body of the clause respectively. The clauses found are pushed onto the trace list (acting as a stack) as terms.

```
solve(X) :-
    solve(X, []).
solve((A, B), Trace) :-
    !,
    solve(A, Trace),
    solve(B, Trace).

solve(not X, Trace) :-
    not solve(X, [not X | Trace]).

solve(Goal, Trace) :-
    system(Goal),
    !,
    call(Goal).

solve(Goal, Trace) :-
    clause(Goal, Body),
    solve(Body, [(Goal :- Body) | Trace]).
```

Figure 3. Interpreter with trace for PROLOG.

It is possible to write a shorter interpreter for PROLOG, e.g. the single clause *solve(X)←call(X)*. The program of Figure 3 captures just sufficient of the control flow to be useful.

### 2.6. The problem of flexibility

It is frequently observed that expert system shells are inflexible. This problem arises from the tension between requiring a program to represent knowledge and to explain itself and also change the representations used by the program and its control structures fairly frequently. It is difficult to design a programming language which can meet both these requirements.

Our view is that this problem is largely one of the distance between the underlying implementation language of the shell, and the representation language interpreted by the shell. In YAPES a radical solution is provided by using a uniform representation language. The shell is written in the language used to represent knowledge, namely Horn clause logic. Our problem then becomes one of modifying the core interpreter (as shown in Figure 1) and extending, if necessary, the meta-logical control language used by the interpreter.

Our experience with YAPES shows that this is not an intractable problem due to the power and flexibility of unification. It does not take long to modify the interpreter simply because the code is compact. The main interpreter for YAPES contains only eight clauses.

It could be argued that the reason for this is that the control structure of YAPES is very similar to that of PROLOG. This does not seem to be the

case. Shapiro (1983a) has published a co-routining interpreter for *Concurrent* PROLOG, written in PROLOG which consists of only 50 lines of code (32 clauses). Equally as important, the very different control structure used by an interpreter such as this does not affect the explanation capabilities in terms of proof trees discussed above, or the extensions to the language used for plausible inference. All this is carried over unchanged: the system is modular.

### 2.7. Implementation

A final practical point should be made at this point about the practicalities of the use of PROLOG. There is little point producing a system, however elegant and simple. if it is impractical to run it in the computing environment of typical users.

Historically PROLOG has suffered from the problem of only being available, in efficient implementations, on a small number of machines—notably the DEC 10/20 range for Edinburgh PROLOG. YAPES has been run successfully with relatively small knowledge-bases (<100 rules) under C-PROLOG on a VAX-750. Recently new versions of PROLOG (e.g. Quintus PROLOG) have become available which have over an order of magnitude gain in speed and space over C-PROLOG. These PROLOGS are available on a wide range of (relatively) cheap hardware and are capable of supporting very large knowledge-bases for YAPES, as well as providing the efficient interface to the underlying operating system which would be necessary in any commercial use of such expert systems.

## 3. THE FACILITIES OF YAPES

The facilities available in YAPES can be split conveniently into three parts.

1. Extensions to the language of Horn clause logic.
2. Interaction with the user.
3. Checking the knowledge-base.

### 3.1. Extensions to the language of Horn clause logic

It is generally recognized that the ability to reason about uncertainty is very useful in an expert system. Unfortunately it is not clear how this should be done.

The first expert system to use plausible reasoning was MYCIN (Shortliffe, 1976) which used a combination of probability theory and the theory of confirmation.

Many attempts have been made in the past decade to improve on the MYCIN formalism, which was recognized to have many defects. Most of these attempts have centred on the problem of assigning probabilities to hypotheses/events. The main issue is one of complexity. Given $N$ events a fully specified probability distribution requires in the order of $2^N$ constraints.

Attempts to apply constraints by making *a priori* assumptions about distributions (e.g. Prospector, Duda *et al.*, 1979) have found that such assumptions are often incorrect and/or unduly restrictive. Attempts such as in Cheeseman (1983) to estimate distributions using entropy measures run into problems of computational complexity.

A further, perhaps more fundamental, problem is that in many circumstances probabilities seen as numbers (or ranges of numbers) are not appropriate in a knowledge-based system—they are too opaque.

In YAPES an attempt is made to solve these problems by shifting our theoretical base and providing a richer domain for certainties than the real number line. We also provide a well-founded semantics that avoids problems of computational complexity.

We proceed by extending the domain of values that literals can take from {*true, false*} to *lattices* of values.

The detailed theory describing how lattices of truth values can be integrated into our logic is described in Appendix A. Here we shall concentrate on the specific system currently used in YAPES. We should emphasize that the system described below is only one of many that could be chosen, and that YAPES is *modular.* A wide variety of inference systems could be quickly implemented.

We have departed from the use of numerical certainties and use what are termed *justifications.* An example of a justification is shown in Figure 4. The literal *entitledToAllowance* (*tim, 52*) is *justified* by the structure below it, which is a justification. This justification is a forest of justifications. The first three are units, the fourth is itself a tree.

Every literal now contains a justification, rather than the (implicit) value *true* as before. The empty justification [] corresponds to *true.*

The use of justifications allows us to control the reasoning of the system more fully. We can specify that two justifications are incompatible for example, and give a low overall certainty to their combination. The

**Tim is entitled to an allowance of fl.52**
is justified if
  **The trip was in Holland**
  and
  **tim didnt go on a boat**
  and
 **The plane cost is more than the car cost in Holland**
  is justified if
   **The trip was in Holland**
 A justification expressed in English. The literal at the top level is *entitled to Allowance* (*tim,* 52)

Figure 4. A justification.

main use at present in YAPES is to reduce the number of questions asked of the user, by making assumptions about the truth of certain literals. If the user dislikes any of the assumptions made these can be overridden by use of the threshold mechanism described below.

We shall now discuss in more detail the calculus of justification currently used by YAPES.

### 3.1.1. *A calculus of justification*

Figure 5 illustrates a (simplified) version of the top-level YAPES interpreter, which calculates certainties. The omissions are intended to make the structure of the mechanism clear while omitting details such as the user interface. A full listing of this interpreter is provided in Appendix B.

The interpreter uses the goal *solve/3* with three arguments. The first argument is the goal to be solved, the second is the *threshold* setting the minimum value for a justification. *Solve* will fail if the goal cannot be solved with at least this certainty. The third argument is the *certainty* of the goal.

A clause is now a pair $\langle just, G \leftarrow B \rangle$ where $G \leftarrow B$ is a Horn clause and *just* is the name of its associated certainty function. The certainty of $G$ is determined from the certainty of $B$ using *just*. The empty

```
solve((A, B), Thresh, Value)
   :- !,
      solve(A, Thresh, Aval),
      solve(B, Thresh, Bval),
      combined_and(Aval, Bval, Value),
      less(Thresh, Value).

solve(not Goal, Thresh, [])
   :- !,
      not solve(Goal, Thresh, _).

solve(Goal, Thresh, [])
   :- system(Goal),
      !,
      call(Goal).

solve(Goal, Thresh, Value)
   :- dataclause(Just, Goal, Body),
      solve(Body, Thresh, BodyVal),
      Combine_if(BodyVal, Just, Value),
      less(Thresh, Value).
```

Figure 5. An interpreter using certainties.

justification (corresponding to truth) is []. Two procedures are used to determine the certainty of $G$. One to determine the certainty of $B$ as a conjunction of goals $B_i$ (combine_and), and one to determine the value of $G$ given a value for $B$ and *just* (combine_if).

The implementation of thresholding imposes a restriction on derivations. Given a threshold $T$ for the top-level goal $G$, every subgoal of $G$ in the proof tree must have a certainty of at least $T$. This reflects the intuitive idea that the certainty of a conclusion cannot be greater than the certainty of any of its premises. In practice this greatly reduces the amount of search necessary to derive the certainty of a goal.

The *default* for certainty functions in the present implementation is to combine the justifications for each of the $B_i$ into a tree with *just* as the root and the certainties of the $B_i$ as subtrees. If the user wishes to override these defaults he/she must write explicit code for the combining functions in any particular case. Similarly a default ordering is imposed on justifications. Given two justifications $A$ and $B$ we say that $A \leq B$ if the set of nodes of $B$ is a subset of the nodes of $A$. The minimal element *any* is also defined such that $any \leq A$ for all $A$.

The user must ensure that the relevant *monotonicity* properties are satisfied when creating defaults. The requirement that the default be monotonic says that if any goals in the body of a clause become more certain then the certainty of the conclusion should not become less likely. This is, on the whole, an intuitively natural condition, the technical reasons for it are described in Appendix A.

*Expressive power*

There are some circumstances where the user's knowledge about a domain suggests that the above monotonicity requirements are not met. An example, using propositional variables, shows such a situation. Let us assume that we have a goal $h$ and two contributing factors $e_1$ and $e_2$. We may know that either $e_1$ or $e_2$ alone support $h$, but that their joint presence does not. If we have a Horn clause rule: $h \leftarrow e_1, e_2$ in our knowledge-base the monotonicity criterion fails to hold, as either $e_1$ or $e_2$ becomes more likely as the likelihood of $h$ decreases.

This type of constraint cannot be directly expressed within our certainty calculus. To see why we can consider the problem of expressing the exclusive or of two goals $e_1$ and $e_2$ in PROLOG. We wish to say that $h$ is true if $e_1$ is true and $e_2$ is false, or if $e_1$ is false and $e_2$ is true. This corresponds to the problem described above, except that we are considering truth-values rather than likelihoods. Horn clause logic, and therefore a PROLOG interpreter cannot express negation directly and must use the idea of 'negation as failure'; a goal is false if it cannot be proved true. The PROLOG solution to the problem of exclusive or would be (using

the unary predicate not to express negation as failure):

$$h \leftarrow e_1,\ not(e_2), \quad h \leftarrow e_2,\ not(e_1).$$

A solution of this form can be used within our certainty calculus. To see how this works we shall first describe its workings from a slightly different viewpoint.

The standard PROLOG interpreter can be understood as producing a proof tree that provides a proof of the top-level query to PROLOG. The value returned by this proof tree is **true**. If no proof tree can be produced the value returned is false. We can view the answer **true** returned as a function of the proof tree, all proof trees for the query will return the same answer. Using the calculus of certainties implemented in YAPES, proof trees can return a wider range of values. Using lattices as the domain of certainties allows us to return the whole proof tree as a certainty if so desired. The problem facing the expert system builder using this formulation is to establish the correct degree of abstraction desired from a proof tree. The implementation described above returns the assumptions needed to produce that particular proof tree. It is an important consequence of this approach that the certainty of a query now depends on the manner in which it was proved. This is not the case for standard Horn clause logic.

Let us reconsider the example presented above. We could write rules related to $h$ as follows:

$$f_1\colon h \leftarrow e_1,\ not(e_2). \quad f_2\colon h \leftarrow e_2,\ not(e_1).$$

The goal $not(e_1)$ in the second rule can now be interpreted in terms of failure to find a proof for $e_1$. In the thresholding interpreter used by YAPES the goal $not(G)$ will be true if $G$ does not have a proof with certainty exceeding the threshold. The use of negation in this manner permits the expression of knowledge about conflicting goals, whilst preserving the semantics of the language.

### 3.2. User interaction in YAPES

The user interaction in YAPES is modelled after the 'query the user' principles described in Sergot (1983). The fundamental idea is to consider the user as a data base from which atomic facts necessary to a proof can be obtained. In effect the PROLOG engine has a choice of data bases from which to obtain facts, the 'system' data base, the 'problem domain' data base and the 'user' data base. This idea can also be applied to any data source other than the user, for example on-line data bases.

#### 3.2.1. *Categories of question*

There are several different categories of question that can be put to the user. There is some overlap between these categories, but in cases where the overlap occurs there is no conflict in the treatment of the questions.

These are:

(a) Questions which have answer false (e.g. 5 is 2+2?);

(b) Ground questions (which have a true/false answer);

(c) Questions which have at most one answer (e.g. sex(tim, *X*));

(d) Questions which may have multiple answers (e.g. is a_son_of(*X*, *Y*)?).

To avoid annoying the user each must be handled in a sensible way. YAPES handles this problem in a way that is intuitively obvious to the user. We shall consider the above categories of question one at a time.

1. Questions which have answer false. These questions are answered by a simple 'no'. It makes no difference whether the question is *ground* (i.e. contains no free variables) or non-ground. The answer to the question is stored in the user data base and will not be asked again, even it the underlying PROLOG inference mechanism backtracks to the question.

2. Questions which have answer true or false because they are ground and about which the system has no meta-level information. The user does not need to supply values for variables in the question. These questions are just asked once and the answer stored.

3. Questions which have a unique answer like 'sex(tim, *S*);' are slightly more complicated in that to handle them properly we need to know something about the relation 'sex(*A*, *B*)'. As far as the logic is concerned any given '*A*' could have any number of sexes (just as any sex in fact has many people!). In the case of the 'sex' relation we need to know that sex is a *function* from persons to sexes, this is achieved by the following meta-level declaration to YAPES:

```
unique(sex(A, B), [A]).
```

This asserts that if variable '*A*' is instantiated then there is a unique '*B*' that makes 'sex(*A*, *B*)' true. When YAPES needs to find out whether 'sex(tim, male)' is true it deduces from the 'unique' assertion above that the most general question to ask is 'sex(tim, *A*)' (which has a unique answer) and asks this question, suitably formatted. Whenever it subsequently has to solve a goal of the form 'sex(tim, *X*)' it has all the necessary information stored in its data base and need not bother the user.

4. Questions which have multiple answers are also dealt with correctly. Let us consider the relation 'isa_son_of(*X*, *Y*)' intended to mean that *X* has son *Y*, which can have several solutions. A PROLOG program using this relation may have to backtrack over several sons before it finds the correct one. Let us consider a specific and rather silly example:

```
chosen_son(Y, X):- %Y has chosen son X
    isa_son_of(Y, X),
    age(X, Age>21),
```

A PROLOG program will typically find an '$X$' satisfying 'chosen_son($Y$, $X$)' (with '$Y$' instantiated) by generating possible solutions with 'isa_son_of($Y$, $X$)' and then the other two goals.

If YAPES has been told that 'isa_son_of($Y$, $X$)' was in the user data base via the meta-level declaration

askable(isa_son_of($Y$, $X$)).

if would query the user for instantiations of $X$. When queried the user has three choices:

1. He/she can type in the name of a son. YAPES will then query for another son (since it has not been told anything about the 'isa_son_of' relation).
2. He/she can type 'no', which should be interpreted as meaning that there are no solutions other than those already provided.
3. He/she can type 'enough', which means: enough questions for now, come back if you need any more solutions later. YAPES will come back when and only when it has backtracked over all the previous user answers and none of them can be used to prove the top-level goal.

### 3.2.2. *Tracing and debugging in YAPES*

The user can ask questions of YAPES while a goal is being evaluated by asking 'why' a question has been put. YAPES then displays a trace of the current path on the proof tree to the user, in a suitably formatted manner.

Two other trace/debugging features are currently implemented in YAPES, the 'how' facility and the 'why-not' facility. The 'how' facility enables the user to see *how* a conclusion was reached. Section 4 contains an example of the use of 'how'. Several implementations of *how* are possible. The one currently used in YAPES allows the user to walk through the proof tree of a successful query.

The problem of diagnosing *why* a query failed is in general more difficult. The reason for this is that for a query to fail every possible proof tree must fail. Not only must we ask why a particular goal failed but must take into account the possibility that earlier goals in the tree produced an incorrect answer, rather than failing.

Without information as to the possible failure modes of a program the only practical possibility seems to be interactive debugging with the user as an oracle, providing information as to which goals should not have failed and which goals have produced an incorrect answer.

In an expert system, however, we do have information as to possible failure modes of a program. We can assume that the knowledge-base has been fully debugged by an expert and that the failure of a goal arises from an answer supplied by the user. The 'why-not' facility of YAPES uses this idea.

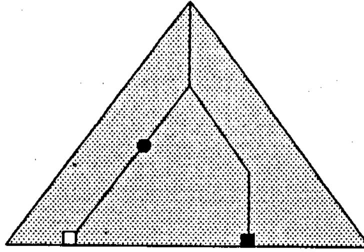

Figure 6. SLD search tree. ■ Node that fails at a user-askable goal. □ Node that fails at a non-user-askable goal. ● Node where a user-askable goal is reduced.

The situation that arises when a goal fails is shown in Figure 6. The SLD-search tree is an *or* tree. Each branch on the tree represents a (partial) proof tree. The tree branches when there is more than one matching clause for a goal. A failing SLD search tree is one where each path through the tree ends in a set of goals that is not satisfiable. In particular if the tree has been produced by PROLOG we know that the leftmost goal is unsatisfiable.

The information that YAPES provides to the user is a failing branch of one of the partial proof trees occurring in the failing SLD-tree. The partial proof tree is selected from a set of candidates in the manner described below, and the branch displayed is that containing the failing goal of the corresponding failed SLD-node.

Candidates are branches of the failed SLD-search tree which:

(a) have as failing goal a user answer; or

(b) have as failing goal a goal which fails to unify because of a substitution occurring at the unification of a user answer with the corresponding atomic goal.

The second condition remainds one of truth-maintenance systems which maintain dependency information about goals (e.g. Doyle, 1979). The difference here is that we are working top-down rather than bottom-up.

There may be many candidates in any given failed SLD-tree. At present YAPES selects a branch that involves the most recent user answer possible. This is a heuristic approach which has worked well so far in practice.

### 3.2.3. *Changing user answers*

Intimately linked with the generation of why-not traces is the ability to change answers to questions. This enables the user to try 'what if' possibilities, and also to correct wrong answers. This facility is currently provided in YAPES.

### 3.3. Type checking user answers

YAPES checks the types of user answers to queries. This facility is based on the PROLOG type checker developed by Mycroft and O'Keefe (1984).

The type checker allows the definition of user-defined types, and the assignment of generic types to procedure arguments.

In practice the user's answer is type checked dynamically and an error message produced if a type violation is detected.

## 4. ANNOTATED TRANSCRIPT OF A YAPES SESSION

The transcript below was taken directly from a terminal session with YAPES. A version of the system with a more sophisticated, window-oriented user interface is also available. All comments are in italics.

*Call YAPES from within PROLOG to start session a knowledge-base has already been loaded.*
|?-*yapes.*

Travel allowance evaluation program

This data base works out whether or not your employee is entitled to an allowance after a trip of duty travelling by car.

Welcome to YAPES

Type help for help

Command: h

HELP INFO
help—This message
go—Go call the knowledge-base
break—Break to PROLOG
change—Change previously recorded facts
clear—Clear the database of all recorded facts
why—Why did a query succeed or fail?
quit—Regretfully leave YAPES

Command: go

What is the name of the employee? tim

Who is the owner of the car? tim

Who gave permission for this trip (mse, da, none)? mse

How many kilometres were driven in total? why

*We are asking why this question has been asked. The answer reveals the goals that are currently being investigated.*

I'm investigating 'The distance travelled was A km.' to establish:
⟨1⟩ The trip was within holland if
  The distance travelled was A km. and
  A < 300.

⟨2⟩ tim had permission to make this trip if
  Permission for the trip was given by mse and
  Authorization by mse is sufficient for trips to holland and
  The trip was within holland.

⟨3⟩ tim is entitled to an allowance of fl.A if
  The car is owned by tim and
  tim had permission to make this trip and
  tim was properly insured and
  The distance travelled was B km. and
  The uncorrected allowance for a distance of B kilometres is fl.C and
  The boat allowance is fl.D and
  The multiplication factor for joint travel is E and
  F is E * (C + D) and
  A plane trip would cost fl.G and
  The maximum of G and F is A.

*Now back to the question*

How many kilometres were driven in total? 100

Does tim have a third-party risk insurance for his car? n

*This is the answer to the query*
tim isnt entitled to a travel allowance

*Now we investigate the failure of the goal*
Command: why

(e.g. why did the goal fail)
Since
You (the user) stated/implied that:
  tim doesnt have third party risk insurance.

Therefore
  tim wasnt properly insured

Therefore
  tim isnt entitled to a travel allowance

*We decide to change a previous answer*
Command: ch

You have told me that:

⟨1⟩ The car is owned by tim
⟨2⟩ The employee's name is tim
⟨3⟩ The distance travelled was 100 km.
⟨4⟩ Permission for the trip was given by mse
⟨5⟩ tim doesnt have third-party risk insurance

*We change two facts (by deleting them from the data base)*
Which facts to be deleted ⟨1.5⟩? 3 5

  Erased

Command: go
*Here is the type checker in operation*
How many kilometres were driven in total? rty

Sorry incorrect type(s) in your answer
The type(s) of The distance travelled was A km. should be:
  integer

How many kilometres were driven in total? 100

Does tim have a third-party risk insurance for his car? y

Who has been driving during the trip? tim

What is the list price of the car? 10000

Did tim travel with colleagues? n

*Now we succeed with the following assumptions made*
Assuming:
⟨1⟩ The trip was in Holland
⟨2⟩ tim didnt go on a boat
⟨3⟩ The plane cost is more than the car cost in Holland
⟨4⟩ The trip was in Holland

then

tim is entitled to an allowance of fl.53
Are any of these assumptions incorrect? y

*We don't like the assumption that the trip was not abroad*
Give me the list of incorrect assumptions: 1

Where did the trip take place (Holland or abroad)? abroad

tim isnt entitled to a travel allowance

Command: why

(e.g. why did the goal fail)

Since
You (the user) stated/implied that:
  not The trip was within holland.

Therefore
  tim didnt have permission to make this trip

Therefore
  tim isnt entitled to a travel allowance

*We change another fact, this time about who gave permission for the trip*
Command: ch

You have told me that:

⟨1⟩ The list price of the car is 10000
⟨2⟩ The car is owned by tim
⟨3⟩ tim was driving
⟨4⟩ The employee's name is tim

⟨5⟩ The trip was within abroad
⟨6⟩ The distance travelled was 100 km.
⟨7⟩ Permission for the trip was given by mse
⟨8⟩ tim has third party risk insurance
⟨9⟩ tim didnt travel with colleagues

Which facts to be deleted ⟨1 · · · 9⟩?7

Erased

Command: go

Who gave permission for this trip (mse, da, none)? da

How much would an airplane ticket have cost for this trip? 200

Assuming:
⟨1⟩ tim didnt go on a boat

then

tim is entitled to an allowance of fl.200

Are any of these assumptions incorrect? n

*Now we are satisfied with the answer we can investigate the solution.*
Command: why
(e.g. why did the goal succeed)
tim is entitled to an allowance of fl.200 since
⟨1⟩ The car is owned by tim
⟨2⟩ tim had permission to make this trip
⟨3⟩ tim was properly insured
⟨4⟩ The distance travelled was 100 km.
⟨5⟩ The uncorrected allowance for a distance of 100 kilometres if fl.52
⟨6⟩ The boat allowance is fl.0
⟨7⟩ The multiplication factor for joint travel is 1
⟨8⟩ 52 is 1 * (52 + 0)
⟨9⟩ A plane trip would cost fl.200
⟨10⟩ The maximum of 200 and 52 is 200

*We can now investigate one of the subgoals*
Which goal (1 to 10)? 2

tim had permission to make this trip since
⟨1⟩ Permission for the trip was given by da
⟨2⟩ Authorization by da is sufficient for trips to abroad
⟨3⟩ The trip was within abroad

Which goal (1 to 3)? 2

*There is a unit clause in the knowledge-base asserting that da can authorise trips abroad*
Authorization by da is sufficient for trips to abroad is a fact

Repeat (r) or stop (s) (tim had permission to make this trip)? s

Repeat (r) or stop (s) (tim is entitled to an allowance of fl.200)? r
tim is entitled to an allowance of fl.200 since
⟨1⟩ The car is owned by tim
⟨2⟩ tim had permission to make this trip
⟨3⟩ tim was properly insured
⟨4⟩ The distance travelled was 100 km.
⟨5⟩ The uncorrected allowance for a distance of 100 kilometres is fl.52
⟨6⟩ The boat allowance is fl.0.
⟨7⟩ The multiplication factor for joint travel is 1
⟨8⟩ 52 is 1 * (52 + 0)
⟨9⟩ A plane trip would cost fl.200
⟨10⟩ The maximum of 200 and 52 is 200

Which goal (1 to 10)? 1

*This is a user-declared fact*
I was told that The car is owned by tim is true

Repeat (r) or stop (s) (tim is entitled to an allowance of fl.200)? s

*Now we quit* YAPES
Command: q

Goodbye—see you again I hope

xxxxxxxxxx

## 5. CONCLUSIONS

YAPES runs under C-PROLOG and Quintus PROLOG on VAX and SUN machines. It is currently used as a teaching tool to demonstrate the use of

logic programming for expert system design. It has demonstrated that:

1. It is feasible to implement the inference engine of an expert system shell in PROLOG. Using Horn clause logic as the knowledge representation language provides a smooth interface with the underlying PROLOG.

2. Explanation facilities can be based on the proof generated by PROLOG as it solves a goal. This proof does not depend on the execution strategy of the interpreter, so that explanation facilities are unaffected by the use of different control regimes. Considering the user as a separate data base of facts allows us to diagnose failure of goals (why not questions) effectively without further interaction with the user.

3. It is useful to interpret plausible inference as an extension to the language of Horn clause logic. In particular we can view plausible inference as concerning the manner in which a goal is proved rather than its truth value *per se*. Further work is necessary to investigate ways in which different proofs of a goal can be manipulated and presented to the user in an informative manner.

**Acknowledgements**

This research was conducted in cooperation with the Shell Research Centre at Sittingbourne Kent, to whom thanks are due for support of the work described and for permission to publish this note.

## APPENDIX A: LOGIC PROGRAMMING WITH UNCERTAINTIES

### 1. Introduction

A recent paper by Shapiro (1983c) introduces the idea of adding uncertainties to logic programs. This idea has been suggested previously: for example Scott and Krauss (1966) develop the semantics of probabilistic logic for countable and infinitary logics. Shapiro's approach is of interest because it provides a simple semantics for Horn clause programs with uncertainties, and is more general than previous approaches—providing a richer set of combining functions for certainties.

This note describes Shapiro's approach and generalizes his results in a number of ways. In particular a fixpoint semantics is given, more general classes of uncertainty functions are allowed and the domain of certainties is extended.

This approach is of particular interest to those involved in the development of expert systems. A large class of such systems use one form or other of uncertain reasoning. Few of these systems are well founded from a theoretical point of view. Logic programs with uncertainties may provide a more suitable framework in which to pursue uncertain reasoning.

## 2. Basic definitions

*Definition.* A certainty space **C** is a complete lattice. This is a partially ordered (under $\leq$) set $C$, with operations $\cup$ and $\cap$ (least upper bound and greatest lower bound respectively) defined for every subset of $C$.
*Definition.* We extend $\leq$ to sequences over $C$ by defining: $\langle c_1, \ldots, c_n \rangle \leq \langle c'_1, \ldots, c'_n \rangle$ iff $c_1 \leq c'_1$ and ... and $c_n \leq c'_n$.
*Definition.* A function **f** from sequences of certainties to certainties in some certainty space **C** is monotone iff for all sequences $s_1$ and $s_2$ of length $n$ over **C**, $s_1 \leq s_2$ implies $f(s_1) \leq f(s_2)$.
*Definition.* A logic program with uncertainties is a finite (non-empty) set **P** of pairs of the form $\langle \mathbf{A} \leftarrow \mathbf{B}, \mathbf{f} \rangle$ where $\mathbf{A} \leftarrow \mathbf{B}$ is a definite (or Horn) clause, and **f** is a montone function from sequences of certainties to certainties.

## 3. Semantics

We are now in a position to define a semantics for logic programs with uncertainties. This semantics is an extension to that given by Van Emden and Kowalski (1976) for definite clause programs without uncertainties. The reader can check that the semantics given here reduces to the semantics of logic programs without uncertainties in the case that our domain of certainties is {**false**, **true**} together with appropriate definitions of the functions **f**.
*Definition.* The Herbrand Universe **U(P)** is defined recursively as,

1. The set of constant symbols in **P** (or the constant symbol **a** if there are none).
2. All atoms of the form $\mathbf{P}(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ where the $\mathbf{t}_i$ are in **U(P)**.

*Definition.* The Herbrand base **H(P)** of a logic program **P** is the set of all ground atoms formed by using predicate symbols from **P** with ground terms from the Herbrand Universe **U(P)**.
*Definition.* An interpretation of a logic program with uncertainties **P** is a function from **H(P)** to a certainty space **C**.
*Definition.* An interpretation $\mathbf{I}_1$ is $\leq$ an interpretation $\mathbf{I}_2$ iff $\mathbf{I}_1(\mathbf{a}) \leq \mathbf{I}_2(\mathbf{a})$ for all $a$ in **H(P)**.
*Definition.* A model **M** of **P** is an interpretation of **P** satisfying the following condition: for any clasue $\langle A \leftarrow B, f \rangle$ in **P** and any ground instance $A' \leftarrow B'_1 \,\&\cdots\&\, B'_n$ of the clause, then $\mathbf{M}(A') \geq f(\mathbf{M}(B'_1), \ldots, \mathbf{M}(B'_n))$.

### 3.1. *Model theoretic semantics*

*Definition.* Given two models $M_1$ and $M_2$ for **P** we define the intersection ($\cap$) of the two models pointwise as $M_1 \cap M_2(a) = M_1(a) \cap M_2(a)$
*Proposition.* Given two models $M$ and $M'$ for **P**, the intersection of the two models $M \cap M'$ is a model.

*Proof.* For any $a \in \mathbf{H}(\mathbf{P})$ if there is a ground instance $a \leftarrow b_1 \ \& \cdots \& \ b_n$ of a clause in $\mathbf{P}$, then the following conditions hold:

$$\text{In } M\text{: } M(a) \geq f(\langle M(b_1), \ldots, M(b_n)\rangle) \geq f(\langle M \cap M'(b_1), \ldots, M \cap M'(b_n)\rangle)$$

$$\text{In } M'\text{: } M'(a) \geq f(\langle M'(b_1), \ldots, M'(b_n)\rangle) \geq f(\langle M \cap M'(b_1), \ldots, M \cap M'(b_n)\rangle).$$

$$\text{As } \mathbf{C} \text{ is a lattice } M \cap M'(a) = M(a) \cap M'(a) \geq f(\langle M \cap M'(b_1), \ldots, M \cap M'(b_n)\rangle)$$

which was to be proved.

*Theorem.* $\mathbf{M}(\mathbf{P}) = \bigcap_{M \text{ a model}} M$ exists and is the least model of $\mathbf{P}$.

*Proof.* The proof is straightforward.

The meaning of the logic program $\mathbf{P}$ is defined to be the least model $\mathbf{M}(\mathbf{P})$ of $\mathbf{P}$

3.2. *Fixpoint semantics*

We now provide a constructive definition of $\mathbf{M}(\mathbf{P})$ by showing it to be the least fixpoint of a function $\mathbf{T_P}$ from interpretations to interpretations.

*Definition.* Let $\mathbf{T_P}(I)$ be defined pointwise on an interpretation $I$ as follows:

$$\mathbf{T_P}(I)(a) = \bigcup_{\lambda \in \Lambda_a} (f_\lambda(I)) \text{ where } \Lambda_a = \{a \leftarrow b \mid a \leftarrow b \textit{ is a ground instance}$$

*of a clause in* $\mathbf{P}\}$ and $f_\lambda(I) = f(I(b_1), \ldots, I(b_n))$ where $\lambda = a \leftarrow b_1 \ \& . \cdots \& \ b_n$.

*Proposition.* $\mathbf{T_P}$ is monotone over the lattice of interpretations, and moreover the lattice of interpretations (with *po*$\leq$) is complete.

*Proposition.* $\mathbf{T_P}$ has a least fixpoint.

*Proof.* Since $\mathbf{T_P}$ is a monotone function on a complete lattice, $\mathbf{T_P}$ has a complete lattice of fixpoints and therefore a least fixpoint.

We can elaborate on this result by proving that $\mathbf{T_P}$ is **continuous** as well as monotone if the underlying certainty functions we use are continuous.

*Definition.* A certain function $\mathbf{f}$ from sequences of certainties to certainties is continuous iff $\mathbf{f}(\bigcup_{x \in X} x) = \bigcup_{x \in X} \mathbf{f}(x)$ for all chains $X$.

*Proposition.* If the underlying cerainty functions $\mathbf{f}$ are continuous then the corresponding functions $\mathbf{f}_\lambda$ are continuous.

*Proposition.* If the underlying certainty functions $\mathbf{f}$ are continuous then $\mathbf{T_P}$ is a continuous function, that is: $\bigcup_{n \in \Pi} \mathbf{T}_P(I_n) = \mathbf{T}_P(\bigcup_{n \in \Pi} I_n)$ for all chains $\{I_n \mid n \in \Pi\}$ of interpretations.

*Proof.*

$$\mathbf{T_P}\left(\bigcup_n I_n\right)(a) = \bigcup_{\lambda \in \Lambda_a} f_\lambda\left(\bigcup_n I_n\right) \quad \text{(by definition of } \mathbf{T_P})$$

$$= \bigcup_\lambda \bigcup_n f_\lambda(I_n) \quad \text{(by continuity of the } f_\lambda)$$

$$= \bigcup_n \bigcup_\lambda f_\lambda(I_n) \quad \text{(by interchangeability of } \textit{lubs})$$

$$= \bigcup_n \mathbf{T_P}(I_n)(a) \quad \text{(by definition of } \mathbf{T_P})$$

This completes our discussion of the semantics of logic programs with uncertainties, and shows that the meaning of a logic program with uncertainties can be approximated computationally.

#### 4. Discussion

##### 4.1. *Relation to Shapiro's work*

We have extended the results presented by Shapiro in three ways.

1. The semantics has been extended to provide a fixpoint semantics for uncertainties. It has been shown that if the underlying certainty functions are continuous the certainties of formulas can be approximated computationally.
2. The class of functions used for evaluating certainties is broader, as they use sequences of certainties as the domain rather than multisets of certainties as used by Shapiro. This allows one to distinguish a given literal in a clause as being more important (less important) than others, which is often the case in practical reasoning.
3. The certainty space used is a complete lattice, rather than the interval (0, 1] used by Shapiro. This is important because it permits us to use uncertainties which are not necessarily 'comparable', by insisting only that certainties are partially ordered rather than totally ordered. This is important in practical reasoning, where a total ordering of certainties often requires an expert to make problematic judgements, unwarranted by experience.

### APPENDIX B: THE TOP LEVEL INTERPRETER FOR YAPES

```
% solve/4 has arguments:
%   *Goal*—Goal to be solved
%   *Trace*—Current goal stack
%   *Thresh*—Threshold value
%   *Value*—the assumptions under which the goal holds solve
```

```
solve ((A, B), Trace, Thresh, Value)
  :- !,
     solve(A, Trace, Thresh, Aval),
     solve(B, Trace, Thresh, Bval),
     combine_and(Aval, Bval, Value),  % Combine assumptions for
     ∵d combo
     less(Thresh, Value).  % Value is 'not less' than *Thresh*
                           %the threshold value
% A→B; C is Prolog's form of if··· then··· then··· else
solve ((A→B; C), Trace, Thresh, Value)
  :- !,
     (solve(A, Trace, Thresh, Aval), !,
     solve(B, Trace, Thresh, Bval),
     combine_and(Aval,Bval, Value)
     ;
     solve(C, Trace, Thresh, Value)
     ).
solve((A; B), Trace, Thresh, Value)
  :- !,
     C
     solve(A, Trace, Thresh, Value)
     ;
     solve(B, Trace, Thresh, Value)
     ).
solve(not Goal, Trace, Thresh, [])
  - !, % This cut is essential.
     not solve(Goal, Trace, Thresh,_).
% System goals are always solved without assumptions (they're correct!)
solve(Goal, Trace, Thresh, [])
  :- system(Goal)
     !,
     call(Goal).

% Reduce the goal *Goal* and incorporate its assumptions
% The reduction is by access to non-system clauses,
% either unit clauses provided interactively by the user or clauses in the
% rulebase provided by the user.
solve(Goal, Trace, Thresh, Value)
  :- dataclause(Just, Goal, Body, Trace),
    solve(Body, [(Just:Goal:- body) | Trace], Thresh, BodyVal),
    combine_if(BodyVal, Just, Value),
    less(Thresh, Value).
```

## REFERENCES

Cheeseman, P. (1983) A method of computing generalised Bayesian probability values for expert systems. *IJCAI-83,* 198–202.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence* **12**(3), 231–272.

Duda, R. *et al.* (1979) Model design in the PROSPECTOR consultant system for mineral exploration. In *Expert systems in the micro-electronic age* (ed. D. Michie) pp. 153–167. Edinburgh University Press, Edinburgh.

Hammond, P. (1983) *APES: A Prolog expert system shell,* Department of Computing, Imperial College, London.

Kowalski, R. (1979) *Logic for problem solving.* North Holland, Amsterdam.

Mycroft, A and O'Keefe R. (1984) A polymorphic type system for Prolog. *Artificial Intelligence* **23**(3) 153–167.

Scott, D. and Krauss, P. (1966) Assigning probabilities to logical formulae. In *Aspects of inductive logic* (eds E. Hintikka and P. Suppes) pp. 219–264. North Holland, Amsterdam.

Sergot, M. (1983) A query-the-user facility for logic programming, *Proc. European Conf. on Integrated Interactive Computing Systems,*(eds P. Degano and E. Sandewall), North Holland, Amsterdam.

Shapiro, E. H. (1983a) *A subset of concurrent Prolog and its interpreter.* TR-003. ICOT, Tokyo.

Shapiro, E. H. (1983b) *Algorithmic program debugging.* MIT Press, Cambridge, Mass.

Shapiro, E. Y. (1983c) Logic programs with uncertainties. *IJCAI-83*, 529–532.

Shortliffe, E. H. (1976) *Computer based medical consultations: MYCIN.* Elsevier, Amsterdam.

Van Emden, M. H. and Kowalski, R. A. (1976) The semantics of predicate logic as a programming language. *J. Association for Computing Machinery* **23**, 733–742.

Weyhrauch, R. (1980) Prologomena to a theory of mechanized formal reasoning. *Artificial Intelligence* **13**, 133–70.