1988

MetaProlog Design and Implementation

Hamid Bacha
Syracuse University

LPRG-TR88-3

MetaProlog Design and Implementation

Hamid Bacha

1988

School of Computer and Information Science
Syracuse University
Suite 4-116, Center for Science and Technology
Syracuse, New York 13244-4100

MetaProlog Design and Implementation[1]

HAMID BACHA
Logic Programming Research Group
School of Computer and Information Science
313 Link Hall - Syracuse Univ.
Syracuse, NY 13210 USA

hamid@logiclab.cis.syr.edu

ABSTRACT

Many researchers in the area of logic programming have recognized the limitations of logic languages such as Prolog and suggested a meta level approach as an alternative . Some of the main drawbacks cited are the control strategy, the presence of a single database, and the ad hoc extensions to the basic logic programming paradigm to allow the dynamic modification of the database. The MetaProlog language, which includes Prolog and some of its metalanguage, deals with some of these problems. In this paper, the design and implementation of MetaProlog are described and the changes to the Warren Abstract Machine (WAM) on which it is based are detailed. Familiarity with the WAM is assumed. This work is part of an ongoing research that was reported on in [bacha87].

1. Introduction

Many researchers in the area of logic programming have recognized the limitations of logic languages such as Prolog and suggested a meta level approach as an alternative (see for example [Gallaire80], [Bowen82], [Bowen85a]). Some of the main drawbacks cited are the control strategy, the presence of a single database, and the ad hoc extensions to the basic logic programming paradigm to allow the dynamic modification of the database. The MetaProlog language, which includes Prolog and some of its metalanguage, deals with some of these problems. While the idea of MetaProlog was suggested as

[1]This work was supported by grant F30602-81-C-0169 from AFOSR and administered by RADC.

early as 1982, only a simulator for a portion of the language existed. The main focus of the research reported on here was an efficient implementation based on a compiled approach using the Warren Abstract Machine technology. As a consequence, the system developed is a fast and efficient incremental compiler with all the features of an interpreter. It is intended for experimenting with real-world large-scale applications instead of being limited to the usual toy problems. Since this work is a continuation of the research reported on in [bacha87], some of the basic concepts and definitions from that paper will be repeated in order to set the context and to make this exposition as self-contained as possible.

Every computational system has some explicit and some implicit aspects. The explicit aspects are those that can be talked about and manipulated directly in the language. The implicit aspects are those provided by the underlying architecture. In logic-based systems, the explicit aspects are represented by relations, functions, and objects named by predicates, functions, and constant symbols. The implicit aspects include the formulas (which can be used but not mentioned), the sets of formulas, the rules of inference, the provability relation ($\vdash$), the proof tree, and the control strategy (see [Rivieres86]). Being able to treat as first class objects those otherwise implicit aspects provides a way to reason about them, pass them around as values of meta level variables, and possibly modify them. Some of the tacit and otherwise inaccessible aspects of the system that MetaProlog tries to make explicit include the provability relation (referred to as **demo**), the sets of relations or procedures (referred to as theories), the sets of clauses making up a procedure (viewpoints), and the proof tree.

This paper will discuss the design and implementation of the MetaProlog compiler. Since the treatment of theories as first-class objects is at the heart of MetaProlog, this topic will be addressed first. A method for handling theories will be outlined. The next issue to be tackled is the WAM based Abstract MetaProlog Engine. Finally, we discuss the explicit handling of proof trees and viewpoints and finish with an example on MetaProlog and meta-level reasoning.

2. Multiple Databases

At the heart of the MetaProlog system is the treatment of databases as first-class objects capable of being passed as arguments to procedures and returned as values of variables. These databases are referred to as theories. Theories are sets of definite clauses. They can be created either by explicitly specifying the clauses they contain (eg. consulting a file), or modifying an existing theory. A new theory is created from an old theory by adding (deleting) some clauses possibly to (from) an existing procedure. The following built-ins are used to create new theories:

```
consult(File, NewTheory).
addto(OldTheory, Clauses, NewTheory)
dropfrom(OldTheory, Clauses, NewTheory)
```

NewTheory is a variable which will be instantiated to an internal representation of a theory. The new theory inherits all the procedures of the old theory except the ones being added or modified of which it receives a fresh copy. Theories are accessed through the built-in **demo(Theory, Goal)** which specifies that the goal **Goal** be proved in theory **Theory**. Since theories are values of variables, they exist only in the environments in which those variables are defined and cease to exist when those variables are no longer accessible. This makes the theories **temporary**. Thus, the only way to have access to the clauses in a temporary theory is to make the variable representing that theory an argument to every goal that may be invoked. If a large number of theories needs to be accessed at any time, all those theories have to be passed as arguments of predicates. This is obviously very awkward. To avoid this problem, we let some theories be permanent. A theory is made permanent and global by assigning a name to it. The built-in **nameof(Theory, TheoryName)** associates the name **TheoryName** to the theory represented by **Theory**. Also, **consult(File, TheoryName)** creates a new theory and makes it permanent by giving it the global name **TheoryName**. The theory now becomes accessible simply by using its name as in **demo(TheoryName, Goal)**. As an example, suppose we have a medical expert system where the medical knowledge is split between clinical (in file1) and pathophysiological (in file2). We can create two permanent theories 'clinical', and 'patho' using:

```
consult(file1, clinical).
consult(file2, patho).
```

These two theories can now be accessed throughout the expert system program through their names. If we need to test some hypotheses during the diagnosis, we can create temporary theories as in:

```
addto(Diagnosis, SomeHypo, NewDiagnosis), demo(NewDiagnosis, SomeGoal).
```

The theory NewDiagnosis may be discarded if the new hypothesis does not prove fruitful. In fact, upon backtracking in case of failure, the new theory is automatically discarded while the old theory may still be available if backtracking leads to an environment where it is accessible. This contrasts with Prolog's assert/retract where the damage done to the database by these two predicates is permanent.

The main problem associated with multiple theories is in accessing procedures that are shared by many of them. Put another way, we want to know which theory inherits which procedure and from which theory. The naive but costly way to solve this problem is to have separate copies of the common procedures in each theory. This is obviously very space inefficient, especially with theories that share most of their procedures, and the time spent doing the copying would be excessive. That is in fact the case most of the time with theories created with the built-in **addto(OldTheory, Clause, NewTheory)** where the new theory inherits all the procedures of the old theory except for the one being modified. The new theory can itself be modified to create yet another theory, which can be used to create another theory, and so on. The old theory should be preserved as is since it is a separate theory in its own right and may be used at any time. So, destroying the old theory in favor of the new one (as in Prolog's assert/retract) is also out of question. What we need is a mechanism that allows fast access to both the parent theory and its descendants (when theory T2 is created from T1, we say that T1 is the parent theory and T2 the child theory). A method that attempts to balance fast access to the clauses in every theory with space efficiency will be outlined. This method is presented in full details in [bacha87]. Other aspects of MetaProlog Theories will also be described.

2.1. Theories as Collections of Viewpoints

Let's recall that a procedure in a logic programming language is a set of clauses with the same predicate name and arity. If we view each clause as a belief about the relation defined by the procedure, we can define a relation to be a set of related beliefs. A collection of beliefs is said to be related if they apply to the same procedure. Given a certain procedure, different clauses of that procedure are made visible in different theories as that procedure is modified with the built-ins **addto** and **dropfrom**. In other words, different subsets of the procedure may be true in different theories. These subsets of procedures can be considered as the **viewpoints** of the different theories about the relation defined by the procedure. So, a **viewpoint** is a set of related beliefs about a certain relation. To recap:

A MetaProlog database is a collection of theories and relations (procedures).
A relation is a collection of beliefs (clauses).
A theory is a collection of viewpoints.
A viewpoint is a subset of the collection of beliefs making up a relation.
A belief is either a fact or a rule.

As an example, suppose we are trying to model the popular family game **CLUE**. Each player has ideas as to who the suspects might be, what weapon may have been used, and in what room the murder may have taken place. We can easily represent each player's ideas about the murder scene by a MetaProlog theory as in figure 1. For simplicity, we assume we have only two players. The theories player1 and player2 contain the two players' viewpoints. The MetaProlog database will then have the relations

theory player1:	theory player2:
suspect(scarlet)	suspect(green)
suspect(white)	suspect(scarlet)
suspect(peacock)	
weapon(knife)	weapon(revolver)
weapon(rope)	weapon(rope)
room(kitchen)	room(study) :- suspect(scarlet)
	room(library)

Figure 1: Viewpoints in MetaProlog Theories

shown in figure 2. As the viewpoints of the players gradually change as the game proceeds, the theories can be modified using **addto** and **dropfrom**. The above definitions will allow each theory to be considered as a separate database for clarity at the conceptual level. Yet, they make it possible for all the theories to be combined into a single database at the implementation level for ease of access and space efficiency.

suspect/1:	weapon/1:	room/1:
suspect(scarlet)	weapon(knife)	room(kitchen)
suspect(white)	weapon(rope)	room(study) :- suspect(scarlet)
suspect(peacock)	weapon(revolver)	room(library)
suspect(green)		

Figure 2: Relations in the MetaProlog Database Corresponding to the viewpoints of Fig. 1

At the implementation level, a viewpoint is a block of indexes pointing to the appropriate clauses of a procedure. In addition, a viewpoint contains a reference to the theory in which it originated, as well as a pointer to the viewpoint immediately preceding it. The collection of viewpoints pertaining to the same procedure are chained together from the latest to the earliest. A pointer in the procedure table points to the last viewpoint for each procedure. This will ensure that the latest viewpoints are more easily accessible than earlier ones.

Theories are implemented as a tree structure. The root node is a distinguished theory called **basetheory**. This theory contains all the system defined predicates. All the other theories are a descendant of this main theory. Each theory is either a leaf node or the root of a subtree having for nodes all its descendants. When a theory T2 is created from a theory T1 (e.g. addto(T1, Clauses, T2)), T2 inherits all the viewpoints of its parent theory T1, except those explicitly modified during the creation of T2. Given a theory and a goal (e.g. demo(Theory, Goal)), the viewpoint corresponding to the clause in goal, if it exists, would be found along the branch between basetheory and the given theory. If the given theory has a viewpoint, the clauses in that viewpoint would be used to match against the goal. If not, we methodically look at its ancestors starting from the father and continuing on until we reach basetheory. A very efficient algorithm for matching theories and viewpoints is presented in [bacha87]. The efficiency of the algorithm is due to the fact that we look only at few theories along a single path of the tree. Also, the internal representation of the theories makes it possible to eliminate from consideration collections of ancestor theories all at the same time.

2.2. Context Switching

Since MetaProlog can accommodate many theories, we need to know, at any time, with respect to which theory a goal is to be proved. In other words, we need to know what the current context is, and how to switch to a different one. When the system is started, the default context is basetheory. The context can be changed either directly through the predicate **setcontext** or indirectly through the predicate **demo**. The command **setcontext(NewContext)** changes the current context to the theory named **NewContext**. This command is usually given from the top level to define the default context the system should be in whenever it is done carrying out the proof of a given goal. The predicate **demo(NewContext, Goal)** is the standard way of temporarily changing contexts. In this particular case, we want to switch from the current context to the context specified by the theory **NewContext**, prove the goal **Goal**, then return to the current context. Note that **setcontext** can be used only with permanent theories, while **demo** may be used with any theory.

At the implementation level, a register (meta_CTR) is used to point to the current theory. This register is saved in the choice points so that old contexts can be restored on backtracking. The next section will provide more details about its use.

2.3. Virtual Theories

Occasionally, one may want to view a collection of theories as a single large theory and prove a goal with respect to it. Creating such a theory requires an enormous amount of work and involves copying all the procedures and merging the similar ones. A **virtual theory** is used instead to simulate the collection of theories. The virtual theory is specified as T1+T2+... +Tn as in demo(T1+T2+T3, Goal). It is viewed as having all the clauses from all the theories in the order they appear (i.e. all clauses from the first theory, followed by all the clauses from the second theory, etc...). The given goal and all its subgoals are proved with respect to the virtual theory. The virtual theory is simulated through the use of an extra pointer in the choice point. This next theory pointer plays a similar role to the next clause pointer in the WAM. While the next clause pointer points to an alternative clause to try if the current one fails, the next theory pointer points to the next theory to try for a viewpoint if all the clauses in the current viewpoint fail. An extra choice point is created for this purpose as explained in the next section.

3. Abstract MetaProlog Engine

The Abstract MetaProlog Engine is based on the Warren Abstract Machine (WAM) [Warren83]. The WAM has been modified to accommodate some of the features of MetaProlog such as multiple theories and virtual theories. The overall memory organization has been changed. Some new instructions have been added and some old ones changed or dropped. Two new registers have been added and the choice points have been expanded.

3.1. Memory Organization

The Abstract MetaProlog Engine memory organization is shown on figure 3. The only major change from the WAM is the merging of the code area and the heap. This is necessary since **addto** and **dropfrom** are backtrackable. Code associated with temporary theories should, in principle, be reclaimed at least on failure. The problem introduced by the presence of the code on the heap is the complication of garbage collection. The benefit though is that now we can reclaim space taken up by both inaccessible structures and code when the heap is compacted. See [Cicekli87] for a garbage collection algorithm where both structures and code are stored on the heap.

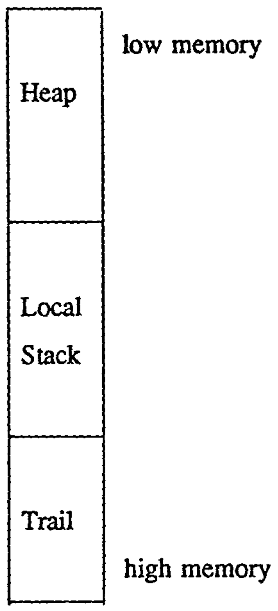

Figure 3: Abstract MetaProlog Engine Memory Organization

3.2. New Instructions and Registers

Two new instructions Fact_Inst and Viewpoint_Inst have been added to help in the matching of theories and their corresponding viewpoints [bacha87]. They are the first instruction in the index blocks that represent viewpoints. All viewpoints occurring in basetheory start with Fact_Inst. Since relations in basetheory are inherited by every theory in the system, they can be seen as facts (in a generalyzed sense of the knowledge: basetheory facts can be either Prolog facts or Prolog rules). Basically, Facts_Inst says do nothing, that is, the current viewpoint is valid in the given theory. On the other hand, Viewpoint_Inst warns that this is a viewpoint that may or may not hold in the given theory and triggers the search for the right viewpoint if it exists. The algorithm described in [bacha87] carries out the matching between theories and their corresponding viewpoints in an efficient manner. Also, the try_me_else type instructions are dropped in favor of the try/retry/trust which are more appropriate for a viewpoint.

One of the new registers needed is the current theory register known as meta_CTR. It points to the theory or virtual theory representing the current context. Since MetaProlog supports multiple theories, it is necessary to know at any time which theory represents the current context. Also, when a failure forces backtracking to a previous context, we must be able to recognize the theory representing that context and restore meta_CTR to it. For this reason, the value of meta_CTR is saved in the choice point any time there are alternative clauses to try. Another register that is used is related to the the handling of the proof trees as first-class objects. The role of this register will be explained in the next section.

While for a given theory there is at most one viewpoint about a given relation, for a virtual theory there may be as many viewpoints as the number of theories making up the virtual theory. All these viewpoints will have to be taken into consideration. For this reason, a new choice point is created whenever there is the possibility of multiple competing viewpoints. The choice points are also expanded to include a pointer to the next theory to be tried should all the clauses in the current viewpoint fail. This pointer plays a similar role to that of the next clause pointer which points to the next clause to try if the current one fails. The only problem is that we now have two different types of choice points. To distinguish between them, the next theory pointer will be set to NIL whenever the next clause pointer is set to an alternative clause within the viewpoint. When the next theory pointer is set to a theory, the next clause pointer will be set to the name of the procedure being invoked. As an example, consider a virtual theory $T_1+T_2+... +T_n$ and a goal G having only one atom with predicate name Pred. Suppose theory T_i ($i<n$) has a viewpoint Vi corresponding to the goal G. The next theory pointer is set to $T_{i+1}+... +T_n$ while the next clause pointer is set to Pred. A new choice point CPi is created. If all the clauses in Vi subsequently fail and backtracking brings us back to CPi, then we have the theories $T_{i+1},..., T_n$ to try for an alternative viewpoint to the procedure Pred indicated by the next clause pointer. To handle backtracking properly, the **Fail** instruction has been modified. If the failure causes backtracking within a viewpoint, the next alternative clause is loaded in the program counter and the next clause pointer is updated to the next to next clause (if any). If the failure causes backtracking to a choice point associated with a virtual theory, the theories pointed to by the next theory register have to be searched for a viewpoint corresponding to the procedure indicated by the next clause pointer. The program counter is set to the first clause in the new viewpoint and the next theory pointer is updated.

4. Proofs as First-Class Objects

One of the main features of expert systems is their ability to explain the conclusions they reach. Since MetaProlog is suitable for expert systems, it should provide some help for automatically gathering the data needed for generating explanations. Proof trees are one of the most useful tools for providing justifications since they include all the subgoals that participate in the evaluation of a given goal. Thus the need to treat proofs as first-class objects to be manipulated very much like any other MetaProlog term. A pleasantly surprising use of proof trees is to affect the control strategy of the system by directing the search for a solution toward a more desirable outcome. Indeed, if the system is presented with a goal and a proof tree of a possible solution, it only needs to check whether there is a proof for the

```
theory info:

flight(C1, C2) :- direct_flight(C1, C2).
flight(C1, C2) :- direct_flight(C1, Ci), flight(Ci, C2).

direct_flight(syracuse, miami).
direct_flight(syracuse, orlando).
direct_flight(syracuse, atlanta).
direct_flight(miami, atlanta).
direct_flight(miami, new_orleans).
direct_flight(orlando, new_orleans).
direct_flight(atlanta, new_orleans).
direct_flight(atlanta, orlando).
```

a) MetaProlog Program

```
demo(info, flight(syracuse,new_orleans), [S1, [direct_flight(syracuse, atlanta)] | S2]).
  Answer1:
  S1 = flight(syracuse,new_orleans)
  S2 = [flight(atlanta,new_orleans),[direct_flight(atlanta,new_orleans]];
  Answer2:
  S1 = flight(syracuse,new_orleans)
  S2 = [flight(atlanta,new_orleans),
             [direct_flight(atlanta,orlando)],
             [flight(orlando,new_orleans),
                   [direct_flight(orlando,new_orleans)] ]];
```

b) Query and Answers

```
1) [flight(syracuse,new_orleans),
     [direct_flight(syracuse,atlanta)],
     [flight(atlanta,new_orleans),
           [direct_flight(atlanta,new_orleans)]]];

2) [flight(syracuse,new_orleans),
     [direct_flight(syracuse,atlanta)],
     [flight(atlanta,new_orleans),
           [direct_flight(atlanta,orlando)],
           [flight(orlando,new_orleans),
                 [direct_flight(orlando,new_orleans)]]]];
```

c) Complete Proof Trees

Figure 4: Proofs as a Control Strategy

given goal along the branches of the search space corresponding to the given proof tree. In other words, the proof tree guides the search for the solution. No wrong or infinite paths need to be followed during the evaluation of the goal. No other solution needs to be considered. In fact, the proof tree can be only partially instantiated. In this case, the system focuses on specific portions of the search space, but is free to explore within these selected subspaces. Early pruning of non-fruitful branches of the search space may lead to a more efficient solution for certain types of problems, despite the overhead associated with the proof trees.

MetaProlog provides such a mechanism through a variation of the **demo** predicate. The two-argument version (demo(Theory, Goal)) evaluates a goal in a given theory. The three-argument version (demo(Theory, Goal, Proof)) evaluates a goal in a given theory and builds the proof that witnesses the derivability of the goal from that theory. The proof is a MetaProlog object (list or structure) which is unified with the third argument of demo. If this argument is a variable, the proof is simply bound to it. However, if it is partially or fully instantiated, it can serve as a control strategy for guiding the proof along a certain path of the search space. As an example of using proofs as a control strategy, assume we have an airline reservation system and we would like to ask whether there is a flight from city C1 to city C2 passing through city Cn. This can be achieved by asking the question about a flight from city C1 to city C2 and partially instantiating the proof tree to contain city Cn as an intermediate city. The example in Figure 4 better illustrates this concept. Assume the proof tree is represented as a list whose head is the main goal and whose tail is a list of subtrees, one for each subgoal. Each subtree is itself a proof tree. We are interested in a flight from Syracuse to New Orleans such that the first stop is in Atlanta. All we need to do is instantiate the head of the first subtree to be a direct flight from Syracuse to Atlanta as shown in Figure 4b. This causes the flights where Miami and Orlando are the first stops to be ignored, thus resulting in an early pruning of the search space. The complete proof trees for the two possible answers are shown in Figure 4c.

Accumulating proofs is a very expensive proposition. The proof tree is made up of structures corresponding to the heads of the clauses that were selected while the proof was being carried out. Since we don't know a priori which clauses are going to succeed, we have to build a structure for every clause attempted. Some structures will then have to be discarded during backtracking. This is indeed a very expensive operation that results in a substantial performance degradation. Clearly, the accumulation of the proofs should be provided as an option that is performed by the system only when needed. One way of achieving this is to provide a mode bit that is examined after the invocation of every goal to check whether a proof structure should be built. However, we feel that even a mode bit is expensive to check so many times. So, for the case of a byte code interpreter, we decided to have two versions of the abstract machine. A fast version that runs without providing any proofs and a slow version that always builds the proof tree. Since the code for the byte code interpreter is very small, the duplication is not a high price to pay compared to the efficiency achieved. Control is switched back and forth between the two machines as goals are evaluated depending on whether a proof is required or not. When a proof is requested control is switched to the slow machine. When the goal that was running with proofs enabled is done, control is returned to the fast machine to continue the execution. A proof tree can be requested several times from within a top level goal that does not require a proof tree. Only the parts requiring the proof tree are executed on the slow machine. The three-argument **demo** predicate can be defined as follows:

```
demo(Theory, Goal, Proof) :-
        set_proof(Proof),
        demo(Theory, Goal),
        reset_proof.
```

The predicate set_proof builds a skeletal structure for the root of the proof tree and unifies it with the argument **Proof**. It then switches control to the slow machine. The goal requiring the proof tree is subsequently called. If this goal succeeds, reset_proof returns control to the fast machine. If it fails, backtracking returns control to the machine that was executing the goal we resume from.

The main problem to contend with is when a failure while evaluating a goal on one machine leads, after backtracking, to a goal that was executing on the other machine. To handle this situation, an extra register is needed. It will be called meta_Proof or proof register. This register is set to NIL when the fast machine has control. When the slow machine is executing, meta_Proof points to the next node of the proof tree. It is also saved in the choice point and reset on backtracking. It is checked after every failure to see which machine should have control. If it is set to NIL, control is passed to (or stays in) the fast machine. If it is not set to NIL, control is passed to (or stays in) the slow machine.

5. Viewpoints as First-Class Objects

Suppose we want to know if two players have the same viewpoint about the relation suspect in the simulation of the game **clue** mentioned earlier. A tedious and inefficient way to accomplish this is to get all the clauses (or beliefs) about the relation in each of the two viewpoints and compare them. An easier and faster way is to confer on the viewpoints the status of first-class objects. Then, the comparison can be accomplished through unification of the viewpoints. Two viewpoints are equivalent if they are unifiable. Since viewpoints are sets of pointers to clauses, two viewpoints are equivalent if they have the same clauses in the same order. Order is important because of the depth-first search strategy. Comparing pointers down at the machine level is much faster than comparing the structures corresponding to MetaProlog clauses. The builtin-ins associated with viewpoints are:

```
viewpoint(Relation, Viewpoint, Theory)
```

```
    belief(Viewpoint, N, Belief)
```

The first built-in takes a predicate name and returns the viewpoint in the current theory corresponding to the relation it names and the ancestor theory the current theory inherited it from (may be the current theory itself). The second built-in takes a viewpoint and an integer N and returns the Nth belief in the viewpoint. If N is larger than the number of clauses in the viewpoint, it just fails. To go back to our example of comparing the viewpoints of player1 and player2, we can use the following:

```
    test(Player1, Player2) :-
            demo(Player1, viewpoint(suspect/1, VP1, T1),
            demo(Player2, viewpoint(suspect/1, VP2, T2),
            (VP1 = VP2 ->
                    write('same viewpoint');
                    write('different viewpoints') ).
```

If the two theories Player1 and Player2 inherited the viewpoint from a common ancestor, T1 and T2 would be the same.

The drawback of the above approach is that it works properly only if no more than one copy of any clause exists. To enforce the existence of a single copy of any given clause, any new clause to be added to the database during the creation of a new viewpoint must be compared to all existing clauses in the same procedure. If it does not match any clause, it is compiled and a pointer to it is used in the viewpoint. Otherwise, the pointer to the existing clause is used in the viewpoint.

6. Meta-level Reasoning

A limited form of meta-level reasoning is supported by the MetaProlog system. Even though the inference mechanism follows a backward chaining regime, the system can be set up to pursue several lines of reasoning at the same time. Recall that the theories are organized in a tree hierarchy. The idea is to consider each branch of the tree as pursuing a certain line of reasoning. Starting from a theory T, we can add some assumptions creating a theory T_{i1}. After doing some work in T_{i1}, we can add some more assumptions creating T_{i2}. We can repeat the same process until we reach a certain theory T_{in}. The branch consisting of the theories from basetheory to T_{in} (basetheory,... T, T_{i1},... T_{in}) is referred to as the current line of reasoning. At any time, we can suspend work along this branch and start a different line of reasoning along another branch, say T, T_{j1},... T_{jm}, using a different set of assumptions. Of course, we can also decide to freeze everything along this new branch and either start a new line of reasoning, or

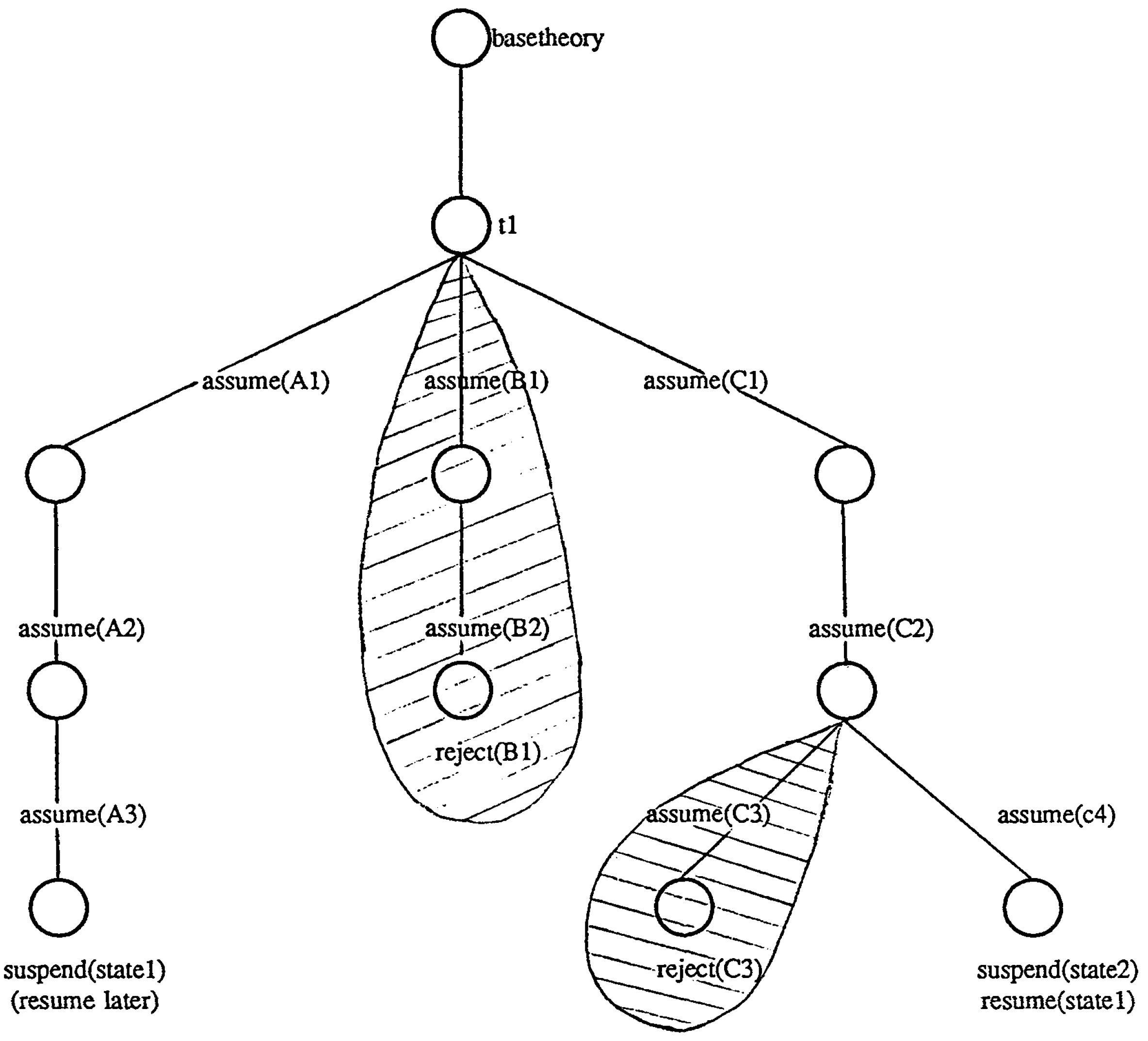

Figure 5: Pursuing Multiple Hypotheses at the Same Time in MetaProlog

go back to a previous one. Sometimes, we may want to abandon a line of reasoning altogether or just retract to a previous position. Four built-ins are provided to support these features: **assume(Belief)**, **reject(Belief)**, **suspend(State)**, and **resume(State)**. The first predicate states that we want to add a certain assumption to the current context, creating a new context. The new context will become the current context. The second predicate states that we want to retract to a previous context, the one just before the specified assumption was added. Any work done between the moment the specified assumption was added and the current context will be trashed. The third predicate assigns the name State (a ground term) to the current line of reasoning and freezes it as is. This command is used when we are ready to explore a new hypothesis along a different line of reasoning but we want to reserve the option of continuing the current one later on. The fourth and final predicate directs the system to go back to a previous line of reasoning suspended under the name State. Note that we can only reject assumptions made along the current line of reasoning. To reject any other assumption, we must first go back to the branch along which it was added using the built-in **resume**. The following example gives an idea on how these features may be used.

```
starting in theory t1
assume(A1),
do some work,
assume(A2),
do some more work,
assume(A3),
do some more work,
difficulties encountered,  % nowhere to go from here for the moment
suspend(state1),           % save what was done so far
setcontext(t1),           % back to where we started from (t1)...
assume(B1),                % ...and try another line of reasoning
do some work,
assume(B2),
do some more work,
major difficulties,        % we want to reconsider
reject(B1),               % abandon current line of reasoning (back to t1)
assume(C1),                % start over
do some work,
assume(C2),
do some more work,
assume(C3),
do some more work,
some problems,
reject(C3),              % back to the context just before last assumption
assume(C4),
do some more work,
nothing promising,
suspend(state2),
resume(state1),            % back to where we left in state1
   ...
```

The tree structure corresponding to this example is show in Figure 5. This example is not really a program, but rather a high level description of the different steps a program may go through. A top level program could recursively add some assumptions, evaluate the results, then decide whether to continue, suspend, retract to a previous position, completely abandon a certain line of reasoning, or resume a previously suspended line of reasoning.

A limited form of belief revision is provided through the use of **assume** and **reject**[2]. After an assumption has been made, we may find out that it is erroneous and decide to remove it as well as all the inferences based on it. Unfortunately, all we can do is go back to a state just prior to when that assumption was made. All inferences made along the line of reasoning containing that assumption between the moment it was added and the context in which it was rejected become inaccessible. The other lines of reasoning are totally unaffected. A true belief system would remove only the inferences associated with the erroneous assumption.

The features just outlined, combined with MetaProlog's support of multiple contexts and fast context switching, should provide excellent tools for writing expert system shells and other kinds of intelligent systems. Though the method of inference is a goal-driven backward chaining mechanism, breadth-first reasoning can still be simulated through the use of multiple lines of reasoning. Many hypotheses can be pursued at the same time, alternating between them through the use of the **suspend** and **resume** mechanisms[3]. Each hypothesis, in turn, is explored using backward chaining.

Acknowledgement:

The author is very grateful to Ken Bowen for his leading role in all aspects of the development of MetaProlog. He also wishes to thank the many people who worked on the system over the years: Kevin Buettner, Ilyas Cicekli, Keith Hughes, Andy Turk, and Toby Weinberg.

[2]This machinery is neutral with regard to the reasons for revision and the goals for this revision. It can be used to implement specific philosophies of belief revision.

[3]Alternatively, a top-level program could create and carry around a list of theories representing a breadth-first development. However, such an approach would not necessarily take advantage of the underlying MetaProlog machinery.

REFERENCES

[Attardi84]
Attardi G., Simi M. (1984): Metalanguage and reasoning across viewpoints. In proc. 6th ECAI, Pisa, Italy, 1984.

[Bacha87]
Bacha, H. (1987): Meta-Level Programming: a Compiled Approach. In proc. of 4th International Conference on Logic Programming, Melbourne, Australia, 1987.

[Bacha87b]
Bacha, H. (1987): MetaProlog User Manual. Logic Programming Research Group Technical Report TR87-10. School of Computer and Information Science, Syracuse University.

[Batali86]
Batali, J. (1986): Reasoning about Control in Software Meta-Level Architectures. In proc. from the conference on Meta-Level Architectures and reflection, Alghero-Sardinia, Italy.

[Barbuti86]
Barbuti, R., Mancarella, P., Pedresch, D., and Turini F. (1986): Intensional Negation in Logic Programs. Submitted for publication to Journal of Logic Programming.

[Bowen82]
Bowen, K.A., Kowalski, R (1982): Amalgamating Language and Metalanguage in Logic Programming. Logic Programming, eds. Clark K.L. and Tarnlund S.A., Academic Press, New York, pp. 153-172.

[Bowen85a]
Bowen, K.A. and Weinber, T. (1985): A Meta-level Extention of Prolog. In proc. of 1985 IEEE Symposium on Logic Programming. Boston. pp. 669-675.

[Bowen85b]
Bowen, K.A.: Meta-level programming and knowledge representation. New Generation Computing, 3(4):359-383, 1985.

[Bowen86]
Bowen, K.A., Buettner, K.A., Cicekli, I., Turk, A. (1986): A Fast Incremental Portable Prolog Compiler. In proc. of 3rd International Conference on Logic Programming, Lodon 1986.

[Cicekli87]:
Cicekli, I. (1987): A Garbage Collector for the MetaProlog System. Logic Programming Research Group Technical Report. School of Computer and Information Science. Syracuse University.

[Gallaire80]
Gallaire, H., Lasserre, C. (1980): A Control Metalanguage for Logic Prgramming. In proc. from Logic Programming Workshop, July 1980.

[Rivieres86]
des Riviere, J. (1986): Meta-Level Facilities in Logic-Based Computational Systems. In proc. from the conference on Meta-Level Architectures and reflection, Alghero-Sardinia, Italy.

[Safra86]
Safra, S., Shapiro, E. (1986): Meta Interpreters For Real. Technical report from the Weizmann Institute in Israel.

[Takeuchi85]
Takeuchi, A., Furukawa, K. (1985): Partial Evaluation of Prolog Programs and its Application to Meta-programming. ICOT Technical Report TR-126.

[Warren83]
Warren, D.H.D., (1983): An Abstract Prolog Instruction Set. Techninal Note 309. Artificial Intelligence Center, SRI International, 1983.