Syracuse University
SURFACE

Electrical Engineering and Computer Science Technical Reports

L.C. Smith College of Engineering and Computer Science

2-1-1987

Meta-level Programming: a Compiled Approach

Hamid Bacha
Syracuse University

Follow this and additional works at: http://surface.syr.edu/eecs_techreports

Part of the Computer Sciences Commons

Recommended Citation
Bacha, Hamid, "Meta-level Programming: a Compiled Approach" (1987). *Electrical Engineering and Computer Science Technical Reports.* Paper 34.
http://surface.syr.edu/eecs_techreports/34

This Report is brought to you for free and open access by the L.C. Smith College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Electrical Engineering and Computer Science Technical Reports by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

Technical Report CIS-87-1
School of Computer and Information Science
Syracuse University

Meta-level Programming: a Compiled Approach*

Hamid Bacha

Logic Programming Research Center
School of Computer and Information Science
Syracuse University
Syracuse, NY 13210 USA
CSNET: bacha@syr

ABSTRACT

There has been some intense research lately focussed on the area of meta-level inference systems. In logic programming, the limitations of Prolog are widely recognized and a meta-level approach has been suggested. Unfortunately, only meta-interpreters have been considered so far. Moreover, these meta-interpreters are often themselves written on top of a Prolog interpreter. These cascaded layers of interpreters result in an enormous slow down, rendering the resulting system practically useless for all but a small number of toy applications. This paper will report on the implementation of a fast incremental metaProlog compiler. In the process, it will explore some of the issues involving meta-level systems in general, and offer the view of a knowledge base as a microcosm of the real world. To this effect, some new definitions are introduced and related to the notion of VIEWPOINT. It is hoped that this type of results will broaden the application area of logic programming to encompass most of the paradigms needed by Artificial Intelligence systems.

February 9, 1987

*This research was supported in part by US Air Force grant F30602-81-C-0169

Meta-level Programming: a Compiled Approach*

Hamid Bacha

Logic Programming Research Center
School of Computer and Information Science
Syracuse University
Syracuse, NY 13210 USA
CSNET: bacha@syr

1. INTRODUCTION

Many researchers in the area of logic programming have recognized the limitations of Prolog and suggested a meta level approach as a more powerful alternative. Unfortunately, only meta-interpreters have been considered so far. Moreover, these meta-interpreters are often themselves written on top of a Prolog interpreter. These cascaded layers of interpreters result in an enormous slow down that renders the resulting system practically useless for all but a small number of toy applications. Under the guidance of Ken Bowen, we have been looking for ways of implementing a metaProlog compiler. This paper reports on some of the results that have been achieved so far. The report will focus on the implementation of a high speed incremental metaProlog compiler based on the Warren Abstract Machine (WAM) instruction set. In addition to running regular Prolog very efficiently, this compiler handles many alternative databases (also referred to as theories) simultaneously, and supports the creation of new theories from existing ones. We hope these results will make logic programming more suitable for a variety of AI applications requiring the use of alternative knowledge bases and frequent fast context switching.

2. OVERVIEW

A meta level approach to logic programming has been advocated by many researchers to overcome some of the limitations of logic based programming languages such as Prolog (see for example [Gallaire80], [Bowen82], [Bowen1-85]). Every computational system has certain aspects that are explicit and can be talked about, while others are implicit and provided by the underlying architecture. The role of meta-level facilities as stated in [Rivieres86] is to make explicit some tacit and otherwise inaccessible aspects of the system. The explicit aspects in logic based systems are represented by relations, functions, and objects named by predicates, functions, and constant symbols. The implicit aspects include the sentences (which can be used but not mentioned), the sets of axioms, the rules of inference, the derivability relation (|- or demo), 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.

All the meta level approaches undertaken so far, to the best of our knowledge, rely on meta-interpreters. What's worse, these interpreters are sometimes themselves written on top of other (Prolog) interpreters. As [Shapiro86] observed, every level of interpretation added to a

*This research was supported in part by US Air Force grant F30602-81-C-0169

computation results in a slow down of an order of magnitude. The use of a very slow system will be restricted to toy applications instead of the more serious investigations that the added meta-level facilities promise to make possible. One remedy that has been suggested to alleviate this problem is a program transformation technique known as partial evaluation [Takeuchi85]. It is claimed that this technique removes unnecessary layers of interpretation by transforming an original program into a new program that inherits the meta-interpreter functionality but runs faster. The main problem we see with this approach is that the resulting program may not have any resemblance to the original program. Being the fallible humans we are, it is hard to write programs that run properly the first time through without a fair amount of debugging. Debugging the resulting program is very hard without a great deal of knowledge about the transformations that have taken place. This makes this method out of reach for the casual user.

To get rid of the meta-interpreters discussed above, we have been looking into ways of implementing a metaProlog compiler. The implicit aspects we have targeted to make explicit are the provability relation (demo in our case) and the set of axioms (theories). We wanted to be able to treat theories as first class objects, and use demo to carry out proofs in any given theory. Since Prolog uses a single theory, we also wanted to be able to execute Prolog programs without any modification. The resulting system we are reporting on is a fast, incremental metaProlog compiler implemented using an augmented Warren Abstract Machine instruction set [Warren83]. It is also based on a modified version of the incremental compiler reported on by Ken Bowen in [Bowen86]. It uses the decompilation technique outlined by Kevin Buettner in [Buettner86] to get back the source code of clauses from compiled code when needed.

3. IMPLEMENTATION CONSIDERATIONS

3.1 MOTIVATIONS FOR CURRENT APPROACH

In logic programming, a procedure is defined as a set of clauses having the same predicate and arity. Prolog has a unique database of clauses (i.e. a single theory), hence every procedure is very well defined and consists of all the clauses with the same predicate and arity in that database. MetaProlog, on the other hand, can have many alternative theories. Although on a conceptual level every theory is a separate database, implementing them that way is very expensive both in time and space. An enormous amount of unnecessary duplication will take place because of the fact that many theories share a lot of procedures. A typical way of creating a new theory in metaProlog is through the add_to predicate as in add_to(OldTheory, Assumption, NewTheory). NewTheory inherits all the procedures defined in OldTheory except for the one involving the clause Assumption. Implementing NewTheory as a separate database means duplicating all the common procedures it shares with OldTheory. The alternative is to view the collection of theories as a single database. While this approach avoids duplication of shared procedures, it introduces some confusion concerning the shared ones. The problem is that the clauses of a procedure visible in one theory are not necessarily the same as the ones visible in another theory. Consider the following procedure:

```
loves(john, jane).
loves(jane, jack).
loves(jane, john).
```

One theory T1 may view the procedure as:

```
loves(john, jane).
loves(jane, john).
```

while another theory, say T2, may view it as:

loves(john, jane).
loves(jane, jack).

Except for their disagreement about this love relationship, T1 and T2 may be identical. That is T1 and T2 have a different *viewpoint* about one relation but agree on everything else. This notion of VIEWPOINT, which is explained below, is going to be the solution to our problem. It will permit us to regard every theory as a virtual database (containing all the procedures defined in that theory) at the conceptual level, yet implement the collection of theories as a single database for efficiency.

3.2 VIEWPOINTS

If we regard a procedure as a relation and its clauses as beliefs about the relation, we can define a relation to be a collection of beliefs the same way a procedure is a collection of clauses. The clauses of a procedure visible in a given theory will be the beliefs about a relation held by that theory, i.e. they constitute the VIEWPOINT of the theory on the relation. From the example above, we can see that the relation being defined is 'love' and that it consists of three possible beliefs. The viewpoint of T1 about this love relation is that John loves Jane and Jane loves John, while that of T2 is that John loves Jane but Jane loves Jack. Since a relation deals with a certain subject (love, in this example), we can as well say that a theory has a viewpoint on the subject. A the conceptual level, we have the following definitions about the portion of our world captured by the database:

A database (or knowledge base) is a collection of theories.
A theory is a collection of VIEWPOINTS.
A VIEWPOINT is a collection of beliefs about a relation (or subject).

We say that a collection of beliefs are related when they define the same relation. Similarly, we say that a collection of VIEWPOINTS are related when they apply to the same relation. So a maximal collection of related VIEWPOINTS represents all the different viewpoints of a relation in our world as captured by the knowledge base. Note that our notion of VIEWPOINTS has nothing to do with that used in the metalanguage OMEGA [Attardi84]. Their viewpoints are collection of assumptions which in our case will be referred to as theories. Our VIEWPOINTS apply to individual relations.

3.3 VIEWPOINTS INHERITANCE

Whenever a new theory is created from an existing one, it inherits all the VIEWPOINTS of its parent theory except for those newly introduced. The newly introduced VIEWPOINTS are either modifications of VIEWPOINTS held by the parent theory, or VIEWPOINTS on relations the parent theory knows nothing about. Since VIEWPOINTS are collections of beliefs, they are modified by either adding some new beliefs (add_to(OldTheory, Beliefs, NewTheory)) or deleting some existing ones (drop_from(OldTheory, Beliefs, NewTheory)). A VIEWPOINT V*i* on a particular relation R held by a theory *Ti* is passed down to all its descendant theories until one of them, say T*j*, develops it own VIEWPOINT V*j* about the relation R. From that point, all the descendants of Tj inherit the new VIEWPOINT V*j*, but the ancestor theories retain their old VIEWPOINT V*i*. (In analogy, if you believe premarital sex is ok but your ancestors don't, you may pass your viewpoint to your children, but you can't influence your ancestor's viewpoint).

We assume the existence of a distinguished theory, call it BASETHEORY, of which every theory is a descendant. Since VIEWPOINTS held by this theory are inherited by all theories (except where modified), we can refer to them as FACTS. So a FACT is a (near) universally held VIEWPOINT. Of course no theory is required to believe what the rest of the world believes and

may develop its own VIEWPOINT on any relation. To determine what VIEWPOINT holds in what theory, each viewpoint carries the signature of the theory in which it originated. A VIEWPOINT V holds in a theory T if it either carries the signature of T, or carries the signature of one of T's ancestors and is the latest such VIEWPOINT about the given relation. In particular, any FACT that has not been disbelieved by one of T's ancestors still holds in T.

It may seem that one may have to examine all the existing theories to decide what VIEWPOINT about a certain relation holds in a given theory. Through a clever representation of theories, we'll show later on an algorithm that retrieves the VIEWPOINTS extremely fast.

4 IMPLEMENTATION STRATEGY

Since the implementation is based on the WAM technology, a slight modification of the instruction set is necessary in order to accommodate metaProlog. In particular, two new registers and two new instructions are needed. The registers are referred to as meta_CTR (Current Theory Register) and meta_BTR (Base Theory Register) and point respectively to a representation of the current theory and BASETHEORY. They are accessed through the builtins current_theory(X) and base_theory(X) respectively. These predicates provide the first link between the object level and the meta level because the (meta) variable in their argument range over sets of clauses from the object level. The new instructions are FACT_INS and VIEWPOINT_INS and will be explained later. The TRY_ME instructions are removed and replaced by the TRY instructions.

From an implementation point of view, the metaProlog database consists of a collection of relations (procedures) and a tree of theories having BASETHEORY as the root. A theory is a collection of VIEWPOINTS. A relation is a collection of beliefs (clauses). A VIEWPOINT is a collection of beliefs from a relation that that are valid in a given theory. Though VIEWPOINTS can be viewed as subsets of relations, the clauses contained in the relations are not duplicated. The VIEWPOINTS contain references to those clauses.

4.1 THEORY REPRESENTATION

The theories are organized as a tree having BASETHEORY as the root and the rest of the theories as nodes. Each theory is assigned a positive integer as its ID number. Theory ID's are integers starting with 0 for BASETHEORY and increasing by one for every new theory. Since they uniquely identify each theory, the theory ID's will sometimes be used to refer to the theories themselves throughout this discussion. Every theory is the root of a subtree having for nodes all its descendants. Figure 1 shows an example of a metaProlog database. BASETHEORY contains the procedures that are common to all the theories. The system builtins, for example, reside in BASETHEORY. New theories are introduced by either consulting an outside file, or modifying and existing theory.

A theory is represented by its ID number, its parent theory, a sequence start theory, and a default theory. The ID number (referred to as THEORYID(T)) and the parent theory (PARENT(T)) of a theory T are exactly what they mean. The sequence start theory ST of a theory T (ST = SEQSTART(T)) is an ancestor theory of T such that the theory ID's of the theories between ST and T form a continuous sequence of consecutive integers. Put another way, for any theory X and integer i = THEORYID(X), if i is between THEORYID(ST) and THEORYID(T), then X is an ancestor of T. The ID of ST must be the smallest possible of any sequence that can be formed. Figure 2 clearly shows the idea of sequence start. The default theory DT of a theory T (DT = DEFAULT(T)) is the oldest ancestor of T other that BASETHEORY. Obviously, a default theory is always a direct descendant of BASETHEORY. The sequence start and the default theory are needed to efficiently locate VIEWPOINTS as will

be seen later on.

Relation p/1:
l1: p(1).
l2: p(2).
l3: p(3).

Relation q/1:
m1: q(a).

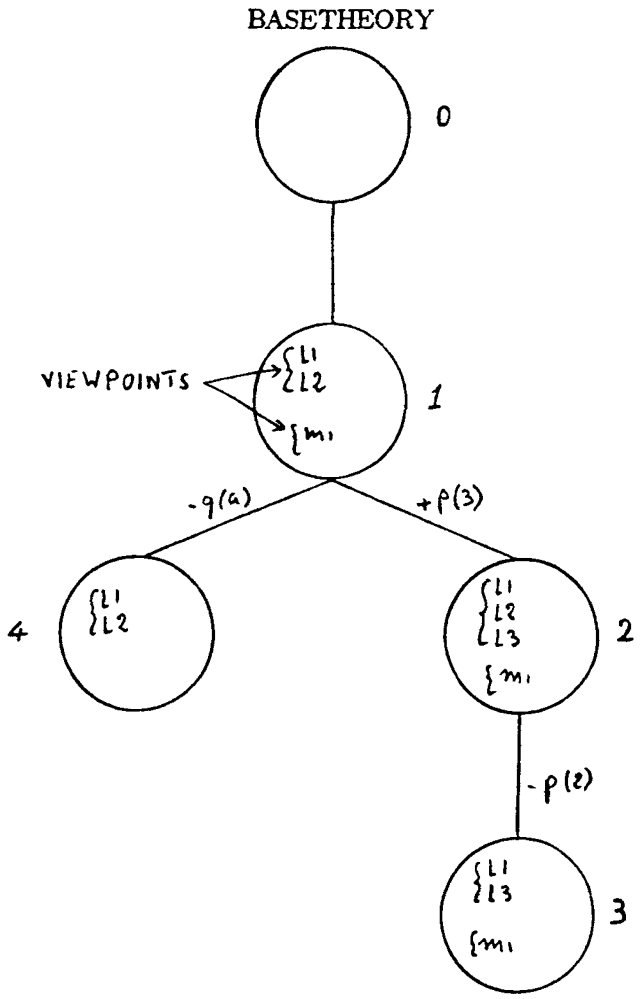

Fig. 1. shows a metaProlog database. + stands for add_to and - for drop_from

4.2 VIEWPOINT REPRESENTATION

At the implementation level, VIEWPOINTS are merely index blocks containing a set of pointers to actual clauses of a certain procedure. We distinguish two levels of indexing: level 1 which contains a chain of TRY instructions, and level 2 which contains the more sophisticated indexing scheme using SWITCH_ON_TERM. In addition, a VIEWPOINT contains the ID of the theory in which it originated (SIGNATURE), and a reference to to the VIEWPOINT (if any) it supersedes (PREV_VP). Preceding the TRY or SWITCH_ON_TERM instruction is either the FACT_INS or VIEWPOINT_INS instruction. FACT_INS indicates that the current VIEWPOINT has been defined in BASETHEORY and can safely be used. VIEWPOINT_INS protects the VIEWPOINT and indicates that a verification must take place to ascertain that the VIEWPOINT

is held by the current theory. A procedure table is used to locate the last VIEWPOINT available for any known relation. When a new VIEWPOINT about an existing relation is created, the entry for that relation in the procedure table is made to point to the new VIEWPOINT, while the latter is made to point to the preceding VIEWPOINT (related VIEWPOINTS are linked together).

4.3 MATCHING VIEWPOINTS AND THEORIES

From our experience with metaProlog, we found out that we usually start with few initial theories (most often one), and create new theories by adding or deleting some clauses. Most of the procedures in the initial theories remain static. That's why we called the initial theories default theories. The default theory of any given theory is the oldest ancestor that is a direct descendant of BASETHEORY (see figure 2). Given a theory and a goal (as in demo(Theory, Goal)), the question we are confronted with is to find in the theory the clauses whose head potentially match the goal. Those clauses are exactly what we refer to as a VIEWPOINT. So the question becomes that of finding the VIEWPOINT (if any) corresponding to the goal that is valid in the theory. If the VIEWPOINT exists, it would have originated either in the given theory or in one of its ancestors. What we need is a fast way to locate that VIEWPOINT. That's where the default theory and the sequence start come into play.

The search for a VIEWPOINT is triggered by the instruction VIEWPOINT_INS. The first place to look for the VIEWPOINT is in the theory we are trying to prove the goal in. The next most likely place is either the default theory, since we expect most of the procedures there to be static, or BASETHEORY. If all fail, we'll have to methodically look at the ancestors of the theory in question. Note that the VIEWPOINTS that occur only in BASETHEORY carry the instruction FACT_INS which doesn't trigger any search. (This feature permits regular Prolog programs to run without any overhead). Since every VIEWPOINT carries the signature (ID number) of the theory in which it originated, we simply check that signature against the ID's of the theories we are matching against. Given a theory T and a goal G, the following algorithm returns the correct VIEWPOINT, if one exists, and NIL otherwise. Assume the goal G has predicate PRED and arity AR.

```
1. LastVP = PROCEDURE_TABLE(PRED,AR);

2. if SIGNATURE(LastVP) = THEORYID(T) or
     SIGNATURE(LastVP) = DEFAULTID(T) or
     SIGNATURE(LastVP) = BASETHEORYID
       return(LastVP);

   while (true) do
3.   if SIGNATURE(LastVP) <= THEORYID(T) and
       SIGNATURE(LastVP) >= SEQSTART(T)
           return LastVP;
4.   if SIGNATURE(LastVP) < SEQSTART(T) or
       (SIGNATURE(LastVP) > THEORYID(T) and LASTVP(LastVP))
           return(NIL);

5.   while SIGNATURE(LastVP) > THEORYID(T) and not LASTVP(LastVP) do
       LastVP = PREV_VP(LastVP);

6.   while SEQSTART(T) > SIGNATURE(LastVP) and
         SEQSTART(T) <> THEORY_ID(meta_BT) do
       T = PARENT(SEQSTART(T));
   end outer while
```

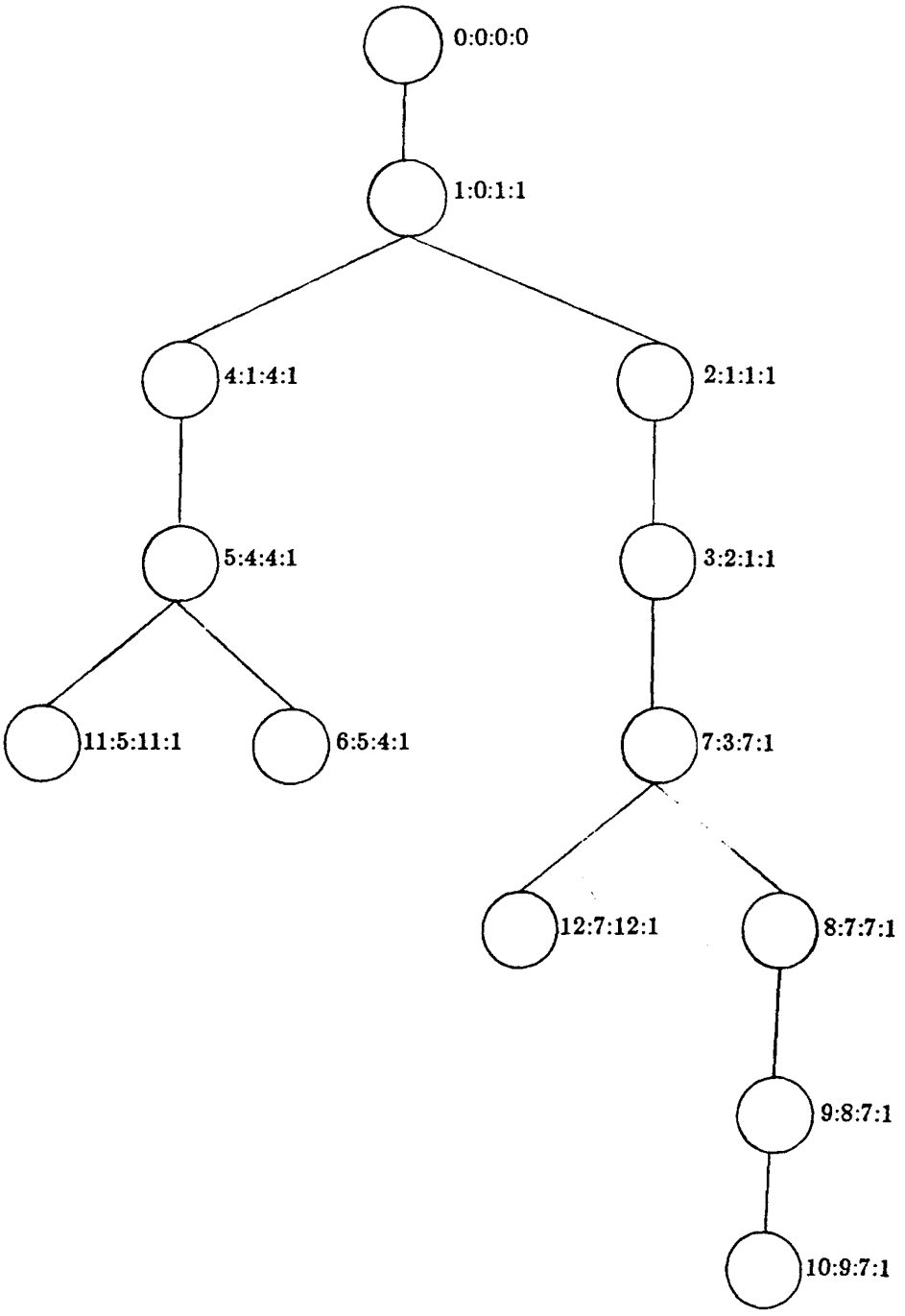

Fig. 2. Each metaProlog theory is represented as 1) theory ID, 2) parent theory, 3) sequence start, 4) default theory

Please note that the algorithm is presented in its most meaningful way rather than its most efficient way. Before we describe the algorithm, we would like to point out that the search for the VIEWPOINT takes place along a single branch of the tree. This branch starts from BASETHEORY and goes all the way to T. It includes all the ancestors of T. What we are trying to do is find the first intersection between the ID's of the theories on this branch and the signatures of a collection of related VIEWPOINTS corresponding to the goal G. If such an intersection exists, we return the corresponding VIEWPOINT. Else, we return NIL. What follows is a description of the algorithm.
DEFAULTID, BASETHEORYID, and SEQSTARTID are the ID's of the default theory, base theory, and sequence start theory respectively. LASTVP(V) is true if V is the last VIEWPOINT in the chain, i.e. there is no VIEWPOINT preceding V.
Statement 1 gets the latest VIEWPOINT from the procedure table.
Statement 2 checks for the most likely cases as explained earlier.
Statement 3 checks whether the VIEWPOINT occurred between the current theory T and the sequence start theory ST. The idea of a sequence of theories can best be illustrated through an example. If we are given theory 10 from example 2 and a possible VIEWPOINT V, we can say that V is valid in T if V originated anywhere in the sequence from 7 to 10. The reason is that V is the latest such VIEWPOINT originating in the sequence (related VIEWPOINTS are chained together through PREV_VP from the latest to the earliest, and we have access to the latest one from the procedure table). So even if we have more than one VIEWPOINT in the sequence, V is still the correct one. Also, if V is the latest VIEWPOINT in the sequence, there is no way for V not to be valid in T because every theory in the sequence is an ancestor of T. On the other hand, if we are given theory 7 and a VIEWPOINT from theory 4, there is no way this VIEWPOINT can be valid in 7 because there is no continuous sequence between 3 and 7. This feature makes the algorithm very fast because we don't need to check every theory in the sequence. We merely ascertain that the VIEWPOINT originated somewhere in between.
Statement 4 checks whether we have enough information to conclude that there is no VIEWPOINT for goal G in theory T. One way to decide that is if the VIEWPOINT on hand originated before T's default theory even existed. This fact can be witnessed by the signature being smaller than the theory ID of the default theory (Remember the smaller the ID the older the theory in terms of creation). Another way is if the signature of the VIEWPOINT on hand is greater than the theory ID of T, and we have no more VIEWPOINTS available.
Statement 5 is reached if the VIEWPOINT on hand is not a suitable one for theory T. We need to find another one that is possibly acceptable. No VIEWPOINT that originated after T was created can be valid in T. So we need to traverse the chain of VIEWPOINTS backward (through PREV_VP) until we find one that originated before T.
Statement 6 traverses the branch of theories backward until we reach either BASETHEORY, or a sequence where the VIEWPOINT from the previous loop may be valid. We know from the previous loop that we have a VIEWPOINT whose signature is less than the ID of T (unless we reached the last available VIEWPOINT), but if that ID is also smaller than SEQSTARTID(T), then we should find an earlier sequence along the branch for a possible match.

An example at this point to clarify the above confusion might be a blessing. Suppose we are solving the goal :-p(X) in theory 10 from example 2 (demo(T10, p(X)). Figure 3 shows three possible VIEWPOINTS for p/1 (we showed the clauses themselves in the VIEWPOINTS instead of pointers to them for clarity). From the procedure table, we get the 3rd VIEWPOINT for p/1. This VIEWPOINT originated in theory 12. Theory 12 is greater then our intended theory 10, so there is no way this VIEWPOINT could be valid in 10. We traverse the chain backward until we find a VIEWPOINT with a signature less than 10. The second VIEWPOINT does the job since it originated in 3. The SEQSTART for 10 is 7 because 7, 8, and 9 are all ancestors of 10. Now 3 is less than 7, so there is no need to consider this sequence. Instead, we go to a previous sequence which happens to be 1,2,3. 3 mathes the signature of the VIEWPOINT, so the second VIEWPOINT is the one that is held by our theory 10. X gets instanciated to b then c. Notice that the next VIEWPOINT originated in 2 which is also an ancestor of 10. But we are only

interested in the first intersection, so it is disregarded.

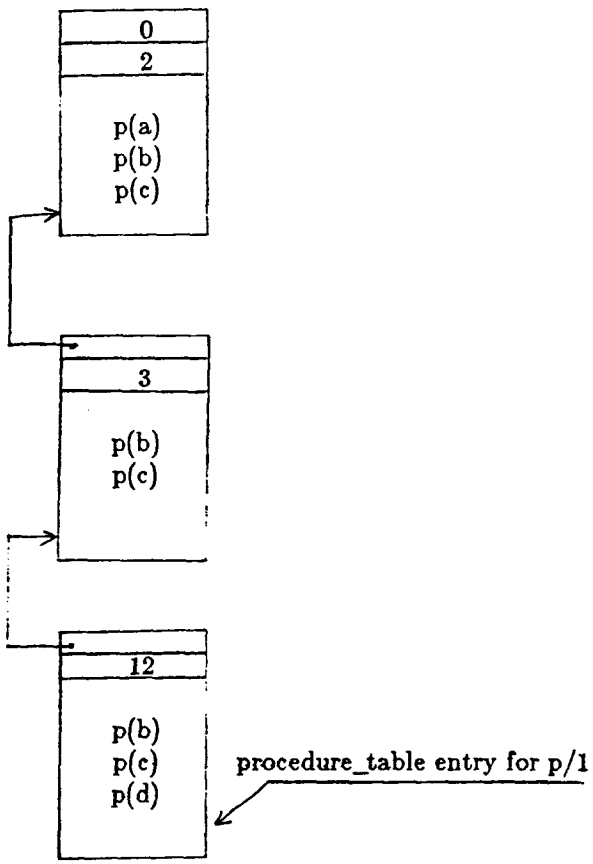

Fig. 3. Three VIEWPOINTS on relation p/1

The algorithm just described is very efficient for 2 reasons. One is that we search only along a single branch (between the given theory and BASETHEORY). For this reason, the total number of theories in the database is immaterial. Second and most important, we don't examine every theory along the branch. We look at sequences of theories at once. So the complexity of the algorithm depends on the number of the sequences along a single branch. Of course, one might create a pathological case where every sequence consists of only one theory. Then we'll have to examine every theory but only along one branch. In our experience, the tendency has been to always create long sequences of theories. The only time we need to create more than one branch is when we have different alternative assumptions to explore. Even then, we tend to create one branch, follow it to a certain depth, then back up to a previous theory along the branch and start on a new branch.

5. PROVABILITY RELATION AND CONTEXT SWITCHING

Once we have theories to play with, the provability relation is easily axiomatized. The demo predicate takes a theory as one argument, and the goal to be proved in that theory as a second argument. (More arguments may be added in the future to handle access to the proof tree, and to affect the control strategy). It becomes the standard way of switching contexts. At a lower level, context switching is achieved via the predicate set_current_theory which takes a theory representation as its argument and sets the register meta_CTR to that value. Demo can be defined as follows:

```
demo(NewTheory, Goal) :-
      current_theory(CT),
      shift_context(NewTheory, CT),
      call(Goal),
      set_current_theory(CT), !.

shift_context(Newtheory, OldTheory) :-
      set_current_theory(NewTheory).
shift_context(Newtheory, OldTheory) :-    /* reset old context if goal fails */
      set_current_theory(OldTheory).
```

Since theories are first class objects and can be values of variables, we added the possibility of using demo with a variable in its first argument (demo(X, Goal)). This is equivalent to saying find me a theory in which the given goal is solved. We intend to add the possibility of distinguishing between the case where any theory in the system can be considered as a solution, and the case where we are interested only in theories along the current line of reasoning. A current line of reasoning is defined as the collection of theories between the current theory and its corresponding default theory. So we can either ask for any theory that is a solution for a given goal, or restrict our attention to theories along the current line of reasoning.

6. REMARKS AND CONCLUSION

The ideas expressed in this paper have already been implemented. The unoptimized experimental version of metaProlog runs the standard naive reverse benchmark for 30 elements on a VAX/780 at about 8K LIPS in BASETHEORY* (C-Prolog gets around 2K LIPS). The overhead is hardly noticeable when the number of sequences along a given branch is small. However, the performance may slow down by as much as 50% for some pathological cases where every sequence contains only one theory. Once again, the number of theories in the database is immaterial. The only theories that are considered during the search for a VIEWPOINT are the ancestors of the given theory, that is the theories on the branch delimited by BASETHEORY and the given theory.

One of the ideas behind this line of research is the view of a knowledge base as a microcosm of the real wold. If we are modeling political decision making for example, we can let the theories encompass the ideas of different decision makers (a person is viewed as a collection of ideas and opinions). Each default theory would have opinions that are widely held in a certain region or culture. BASETHEORY would have the knowledge about policies that are internationally accepted (international law). Other areas such as medical diagnosis or electronics can also be structured to fit within this paradigm.

Programs in BASETHEORY run without any of the overhead associated with metaProlog theories.

The aspects to be made explicit targeted in this first part of the project are the treatment of theories, fast context switching, and the provability relation (demo). Some other aspects to be considered for the future include access to the proof tree, choice of control strategy, and operators on theories. Also in consideration is the addition of control statements about other statements. For some theories, one would like to to have statements about applicable inference steps, together with the relative likelihood that the inference would lead to a desired solution [Batali86]. One promising approach for operators on theories is intentional negation [Barbuti86]. Intentional negation is used to derive the effective complement of a theory which can in turn be used to derive negative information. A program will be represented by two theories, one for computing positive information, the other for negative information. Intentional negation uses SLDIN resolution which has been proved sound and complete [Barbuti86]. To combine two theories, we would combine the positive parts, and synthesize the negative part. To compute the intersection, we combine the negative parts and synthesize the positive part. As stated in [Barbuti86], a set of operators on theories along with constructs for composing them would provide a sound, formal tool for manipulating chunks of knowledge represented as logic theories.

The intense level of research that has been focussed lately on meta-level inference engines seems to indicate that it is a promising area which may yield some very important results in the area of artificial intelligence. Ken Bowen, for example, has shown that many of the common techniques used for knowledge representation in AI can be closely captured in metaPROLOG [Bowen2-85]. John Batali said: 'meta-level architectures provide a technical mean, if not a full account, of how we make our programs understand themselves and exploit that understanding in their operation' [Batali86]. Meta-level systems seem to embody to a certain degree some level of introspection. Introspection plays an important role in awareness and mental activity among humans. This connection alone is enough to encourage us to look further into the nature and architectures of this systems and exploit their full potential toward a new way of computing and computer design.

6. ACKNOWLEDGEMENTS

The author is deeply indebted to Ken Bowen for his role in guiding this research. He is also very grateful to the following people from the Logic Programming Research Group for numerous valuable discussions on metaProlog: Kevin Buettner, Ilyas Cicekli, Andrew Turk, and Keith Hughes.

REFERENCES

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

[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.

[Bowen1-85] 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.

[Bowen2-85] Kenneth Bowen. 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.

[Buettner86] Buettner, K.A. (1986): Fast Decompilation of Compiled Prolog Clauses. In proc. of 3rd International Conference on Logic Programming, Lodon 1986.

[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.

[Shapiro86] Safra, S., Shapiro, E. (1986): Meta Interpreters For Real.

[Takeuchi85] Takeuchi, A., Furukawa, K. (1985): Partial Evaluation of Prolog Programs and its Application to Meta-programming. ICOT Technical Report TR-126. To appear in proc. IFIPS-86.

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