NORTH-HOLLAND

ABSTRACT METAPROLOG ENGINE

ILYAS CICEKLI

▷ A compiler-based meta-level system for MetaProlog language is presented. Since MetaProlog is a meta-level extension of Prolog, the Warren Abstract Machine (WAM) is extended to get an efficient implementation of meta-level facilities; this extension is called the Abstract MetaProlog Engine (AMPE). Since theories and proofs are main meta-level objects in MetaProlog, we discuss their representations and implementations in detail. First, we describe how to efficiently represent theories and derivability relations. At the same time, we present the core part of the AMPE, which supports multiple theories and a fast context switching among theories in the MetaProlog system. Then we describe how to compute proofs, how to shrink the search space of a goal using partially instantiated proofs, and how to represent other control knowledge in a WAM-based system. In addition to computing proofs that are just success branches of search trees, fail branches can also be computed and used in the reasoning process. © Elsevier Science Inc., 1998 ◁

1. INTRODUCTION

Meta-level facilities in logic programming languages provide explicit representation of databases (theories), statements (clauses), derivability relationships between theories and goals, and proofs. These facilities may also include explicit representation of the control knowledge used by the underlying theorem prover. This explicit representation of meta-level objects and control knowledge may improve the expressive power of the language and help to shrink the search space of a goal by avoiding unnecessary searches.

Address correspondence to Ilyas Cicekli, Department of Computer Engineering and Information Science, Bilkent University, 06533 Bilkent, Ankara, Turkey, Email: ilyas@cs.bilkent.edu.tr.
Received October 1995; revised April 1996; accepted January 1997.

655 Avenue of the Americas, New York, NY 10010
0743-1066/98/$19.00
PII S0743-1066(97)00075-7

Many systems with some kind of meta-level facility are presented in the literature [32]. Weyhrauch's FOL system [42, 43] builds up contexts (theories) by declaring predicates, functions, constants, and variables, and defining axioms. In that system, theorems are proved with respect to the axioms of a context and proofs are recorded. In the OMEGA system [3], a metalanguage defines the syntax of expressions and statements, viewpoints describe sets of assumptions, and the consequence concept formalizes the derivability relationship between statements and viewpoints. METALOG [17], an extension of Prolog, explicitly asserts control knowledge separate from regular clauses in the system, and uses this control knowledge when it chooses a literal from a goal list and a clause from the clause database. Russell's MRS system [34] and the system developed by Gallaire and Lasserre [20] also use some kind of explicitly represented control knowledge. The system developed by Lamma et al. [22–24] for the contextual logic programming [27] represents a set of Prolog clauses as a unit, and an ordered set of units as a context. Nadathur et al. [30] create a new context, adding clauses in an implication goal to the current context in their system. Some other researchers in the logic programming community have sought meta-level facilities in meta-interpreters [11, 35, 37–39, 44] based on Prolog. Even standard Prolog [10, 16, 36] has some meta-level facilities. The predicates *assert* and *retract* add and remove clauses from a system-wide database by destroying the old version of that database. The meta predicate *call* tries to prove an explicitly given goal with respect to the single system-wide database. There are no notions of contexts in standard Prolog.

MetaProlog is a meta-level extension of Prolog that has evolved from the research of Bowen and Kowalski [6, 7]. In MetaProlog, theories are made explicit so that they can be manipulated like other data objects in the system. Once theories are made explicit, deductions are made from these theories instead of a single system-wide database. The basic two-argument *demo* predicate in MetaProlog is used to represent the derivability relation between an explicitly represented theory and goal. Another meta-level facility in MetaProlog is dynamically constructed proof trees. They are collected by the system when a goal is proved with respect to a theory by using the three-argument version of a *demo* predicate. A given partially instantiated proof of a goal when the deduction of that goal is started may shrink the search space of that goal.

The derivability relation in the system proposed by Bowen and Kowalski is represented by a two-argument predicate *demo* between theories and goals. The correctness and completeness of the derivability relation are expressed by the reflection rule. For a theory T and a goal G, the reflection rule is as follows:

demo(T, G) iff G is derivable from T

Later, this rule provided the justification for the implementation of the derivability relation as context switches in the MetaProlog system.

The Abstract MetaProlog Engine (AMPE) [14, 15], which efficiently implements meta-level facilities in MetaProlog, is an extension of the Warren Abstract Machine (WAM) [1, 41]. This system provides mechanisms to represent and compute control knowledge and meta-level facilities such as theories, proofs, fail branches, and derivability relation. The AMPE runs in two different modes. The *simple mode* of the AMPE supports multiple theories and the basic two-argument *demo* predicate. In the *proof mode* of the AMPE, proofs and fail branches can be computed and used to control the underlying theorem prover.

There can be many applications of meta-level facilities in a logic programming language. An obvious application of proofs is the explanation facility of an expert system. Collected proofs can be used to give justification about the behavior of a rule-based expert system. Sterling describes a meta-level architecture for expert systems in [40]. In [18], Eshghi shows how to use meta-level knowledge in a fault-finding problem in logic circuits. Cicekli [15] shows how to use multiple theories and fail branches in MetaProlog to express digital circuits and a fault diagnosis algorithm on them. Bowen [8] describes how to use meta-level programming techniques in knowledge representation.

This paper presents the design and implementation of a compiler-based system for the MetaProlog programming language. The MetaProlog system presented here provides efficient implementations of meta-level facilities in MetaProlog, such as theories, proofs, fail branches, and derivability relations. The following four sections are reserved to explain the Abstract Meta Prolog Engine (AMPE) extended from the WAM to get the efficient implementation of MetaProlog. Section 2 describes how multiple theories and the two-argument derivability predicate *demo* are represented in the MetaProlog system, and Section 3 describes the core part of the AMPE, which supports multiple theories in MetaProlog. The core part is used in both modes of the AMPE. Section 4 describes the representation of proofs and fail branches, which are the second most important meta-level objects after theories in the MetaProlog system. Section 5 describes the proof mode of the AMPE, which supports proofs and fail branches in the system. Finally, Section 6 compares our system with other WAM-based systems [22–24, 30] dealing with contexts.

2. METAPROLOG THEORIES

Theories are the first meta-level objects to be addressed in many meta-level systems. They are made explicit in these meta-level systems so that they can be manipulated like other data objects. Since they are explicitly represented, we can reason about them or we can discuss their characteristics. Since explicit representations of theories and statements are available, the provability relation between them can also be explicitly defined.

In the MetaProlog system, theories represent sets of Prolog clauses. There can be more than one theory in the system at any given time. A theory is created from an old theory, and it is discarded when the need for that theory disappears. In fact, when a theory is created, a variable is bound to its internal representation, and that theory is only accessible in contexts where that variable is accessible. They are treated in the same way as any other data structure in the system.

The provability relation holding between a theory and a goal is represented by a two-place demonstrate predicate *demo* in the MetaProlog system. The relation *demo*(*Theory*, *Goal*) holds precisely when *Goal* is provable in *Theory*.

Similar facilities for dealing with theories can also be found in the OMEGA description system [3]. In the OMEGA language, viewpoints describe sets of assumptions, and the consequence concept represents the derivability relationship holding between viewpoints and statements.

Furukawa et al. [28] also use theories to represent logic databases, and they implement the derivability relation between these logic databases and statements.

Worlds in Nakashima's Prolog/KR system [29] are also very similar to theories in the MetaProlog system.

Lamma et al. [22, 23] use contexts to represent multiple theories in a logic programming framework. They also extent the Warren Abstract Machine to the Context Warren Abstract (C-WAM) to handle multiple theories. In their system, a context is created for the derivation of a goal, and it is automatically destroyed after the derivation if the goal is deterministic. Similarly, a context in the system of Nadathur et al. [30] is also created for the derivation of a goal, and it is discarded afterward.

In the rest of this section, we will discuss how to create theories in the MetaProlog system. We will also introduce a mechanism for representing theories to provide fast access to predicates in theories. Another representation of theories is also suggested in [4] and [5].

2.1. Creation of Theories

In Prolog there is only one theory, and all goals are proved with respect to this single theory. On the other hand, there can be more than one theory in MetaProlog at a certain time, so that a goal can be proved with respect to one or more of them. When the predicate *demo*(*Theory*, *Goal*) is submitted as a goal, the MetaProlog system tries to prove *Goal* with respect to *Theory*. The same goal can also be proved with respect to a different theory in the system.

Since there is a single implicitly represented database in Prolog, ad hoc methods are used when there is a need to update this database. The built-in predicates *assert* and *retract* update the Prolog database to create a new version of this database by destroying the old version in the favor of the new version. On the other hand, we do not need to destroy an old theory when we create a new one from that theory in the MetaProlog system.

Theories of the MetaProlog system are organized in a tree whose root is a distinguished theory, the *base theory*. The base theory consists of all built-in predicates, and all other theories in the system are its descendants; i.e., all built-in predicates in the base theory can be accessed from all other theories in the system.

In Figure 1, theories T1 and T2 are created from the base theory. These theories inherit all predicates of the base theory. Similarly, theories T3 and T4 are descendants of the theory T2. Although the arrows in Figure 1 represent the father relation between theories, the father relation is not used in the actual implementation of theories. Instead of the father relation, the *default theory relation* between theories is used in the representation of theories. The default theory relation will be explained in Section 2.3.

A new theory is created from an old theory by adding or dropping some clauses. The new theory inherits all of the procedures of the old theory, except for procedures explicitly modified during its creation. The system can still access both the new theory and the old theory. The following built-in predicates are used to create new theories in the MetaProlog system:

addto(*OldTheory*, *Clauses*, *NewTheory*)

dropfrom(*OldTheory*, *Clauses*, *NewTheory*)

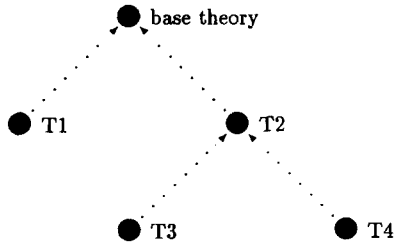


FIGURE 1. A theory tree in the MetaProlog system.

The given clauses (*Clauses*) are added to (dropped from) the given old theory (*OldTheory*) to create a new theory (*NewTheory*) by the predicate *addto* (*dropfrom*). The variable *NewTheory* is bound to the internal representation of the new theory after the execution of one of these commands. Let us assume that *p* is a procedure in *NewTheory*. The clauses of *p* are exactly the same as clauses of *p* in *OldTheory*, if *p* does not contain any clause in *Clauses*. Otherwise, the clauses of *p* in *NewTheory* consist of the clauses in *OldTheory* and *Clauses*, which belong to *p* if *NewTheory* is created by the *addto* predicate. If *NewTheory* is created by the *dropfrom* predicate, the clauses of *p* contain all clauses of *p* in *OldTheory* except the clauses that appear in *Clauses*.

The first argument of the *addto* (*dropfrom*) predicate is a theory (a theory name or a variable bound to the internal representation of a theory), the second argument is a list of clauses, and the third argument must be an unbound variable that is going to be bound to the internal representation of the new theory after the successful execution of the *addto* (*dropfrom*) predicate. Both predicates create a completely new theory with a unique theory identifier in its internal representation. This means that any two theories with two different internal representations are not unifiable in our system, even though they may contain exactly the same clauses. In fact, this is the reason why the last argument of these predicates must be an unbound variable. Two theories can be unifiable only if they have the same internal representations.

2.2. Permanent Theories

After a new theory is created in the MetaProlog system from an old theory that already exists in the system, normally a variable is bound to the internal representation of the new theory. We can access the new theory by using this variable, and this variable should be passed to places where that theory must be accessed. Sometimes, passing this variable to many places is not very practical. For this reason, some theories in the MetaProlog system are given global names, and they can be referred to by using their names anywhere in the program. These theories are called *permanent theories*.

Permanent theories are always present in the system, and they can be accessed via their names. On the other hand, a *temporary theory*, a theory without any name, is only accessible in the environments where there exists at least one variable bound to its internal representation. A *temporary variable* is accessible in the MetaProlog system as long as there is a variable bound to its internal representations. Although the space occupied by a *permanent theory* cannot be reclaimed by the system, the space occupied by a *temporary theory* may be reclaimed during

backtracking, or by the garbage collector if that theory is no longer accessible. In fact, the life cycle of a *temporary theory* is similar to the life cycle of a structure in the heap.

A *permanent theory* can be created by using the built-in predicate *consult*, or a *temporary theory* can be converted into a *permanent theory* by using the built-in predicate *nameof*. For example, when the goal *consult*(*FileName*, *TheoryName*) is executed, a new theory that contains all predicates in the given file is created, and the name *TheoryName* is assigned to it. The built-in *nameof*(*Theory*, *TheoryName*) can convert the temporary theory designated by *Theory* into a permanent theory, and the name *TheoryName* is assigned to it. Afterward, this permanent theory can be accessed via its name at any time.

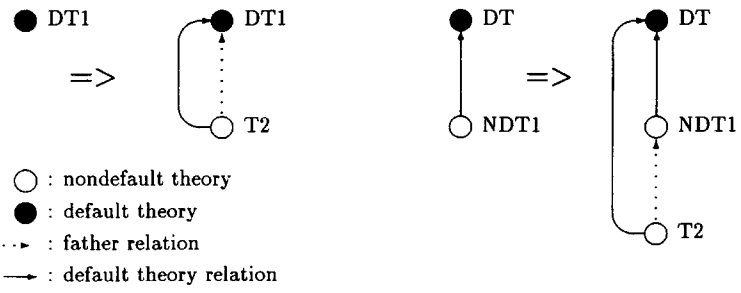
2.3. Default and Nondefault Theories

Theories in the MetaProlog system are classified into two groups: *default theories* and *nondefault theories*. Every theory in the MetaProlog system possesses a default theory, except for the base theory. The default theory of a theory is the theory in which we search for a procedure if the given theory does not know anything about that procedure. The search starts from a node of the theory tree and proceeds with default theories along a certain branch until the procedure is found or the root of the tree is reached.

A *nondefault theory* is a theory that carries complete information about all procedures that are modified in all nondefault ancestor theories between this theory and its default theory. Access to these procedures is very fast, at the expense of copying pointers to these procedures during the creation of a theory from a nondefault theory. Each theory T (default or nondefault) has a pointer to its default theory DT, which happens to be the first ancestor default theory working from T. The descendants of a default theory D do not carry any information about the procedures occurring in D. In other words, the default theory D stops any further propagation of information about procedures from its ancestors to its descendants. If only default theories are used, access to a given procedure in a given theory may require a search through all of its ancestor theories. In this case, access to a procedure may be slow, but no copying of references is needed. Depending on the problem, the system tries to use one or the other approach, or a combination of both to achieve a balance between the speed of access and the space overhead.

When a new theory T is created from a nondefault theory N, the default theory of T will be its father's default theory; i.e., T's default theory will be N's default theory. But if a new theory is created from a default theory, its default theory will be its father. In the first case, the new theory will be at its father's level in the tree. In the second case, the new theory will be at one level above its father's level. Thus we do not need to increment the depth of the tree when a new theory is created from a nondefault theory. In Figure 2a, a theory T2 is created from a default theory DT1. The father of T2 and the default theory of T2 are the same theory. On the other hand, when a theory T2 is created from a nondefault theory NDT1 (cf. Figure 2b), the father of T2 is different from the default theory of T2.

The *naive approach* is to have all of the theories in the system be default theories. In this case, the default theory relation between theories is the same as



a. From A Default Theory b. From A Nondefault Theory

FIGURE 2. Creation of a theory from default and nondefault theories.

the father relation[1] between them. This situation can be seen in Figure 3. In this approach, when a new theory is created, it will be at one level above its father's level, because all theories are default theories. Thus the theory tree can become very deep, which explains why a search for a procedure can be expensive. For example, to reach the procedure "p" from theory T4, it is first searched for in theory T4, where it does not exist. Then it is searched for in theory T2, and finally it is found in theory T1. Thus, to access the procedure "p" from theory T4, we have to search for it in three theories.

To shorten the depth of the theory tree, we introduce nondefault theories. If there is at least one nondefault theory in the system, this situation is called the *nondefault theory approach.* In this approach, the default theory of a given theory can be one of its remote ancestors instead of its father. In fact, when a new theory T2 is created from a nondefault theory T1, the father of T2 will be different from the default theory of T2. If no theory is created from any nondefault theory, the theory tree will be the same as the theory tree in the naive approach. The advantage of the nondefault theory approach is apparent when we start to create theories from nondefault theories. At that time, the depth of the tree will not grow fast, and the search for procedures will generally be shorter.

Figure 4 shows the tree of theories of Figure 3 in the nondefault theory approach. We assume that the only default theory is the base theory, and theories T1, T2, T3, and T4 are nondefault theories. In this tree, reaching a procedure is much faster than reaching the same procedure in the tree of the naive approach. For example, to access the procedure "p" from theory T4, it is sufficient to search only T4, since a pointer to the procedure "p" was copied into T2 and T4 during their creations. On the other hand, in the naive approach we had to search three theories to access the same procedure.

The price we pay for fast access to procedures in the nondefault theory approach is that we have to copy references to procedures of nondefault theories into their descendants. However, since we copy only a reference for each procedure into the new theory, this copying operation does not cost too much.

[1] Internally, there is no father relation in the system. Only the default theory relation between theories is present.

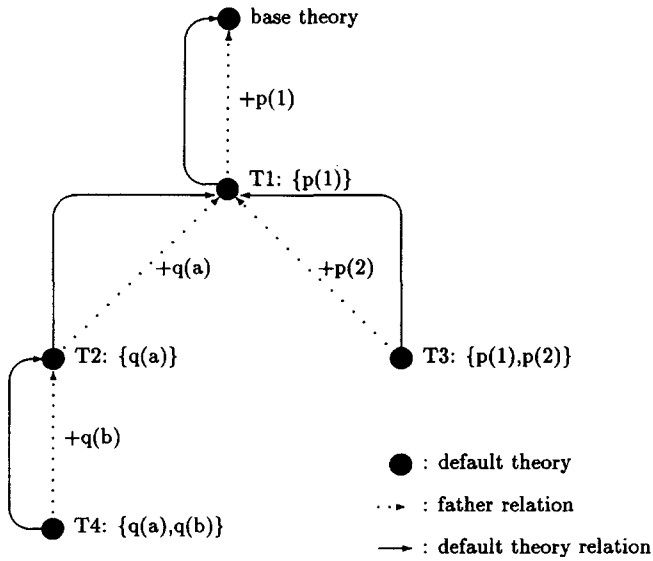


FIGURE 3. A theory tree in the naive approach.

The system should decide which theories ought to be default theories and which ones ought to be nondefault theories. To get the best performance from this approach, the theories with many procedures should be default theories, and the theories with few procedures should be nondefault theories. The decision of the system depends on this observation. The system assumes that a theory T is a

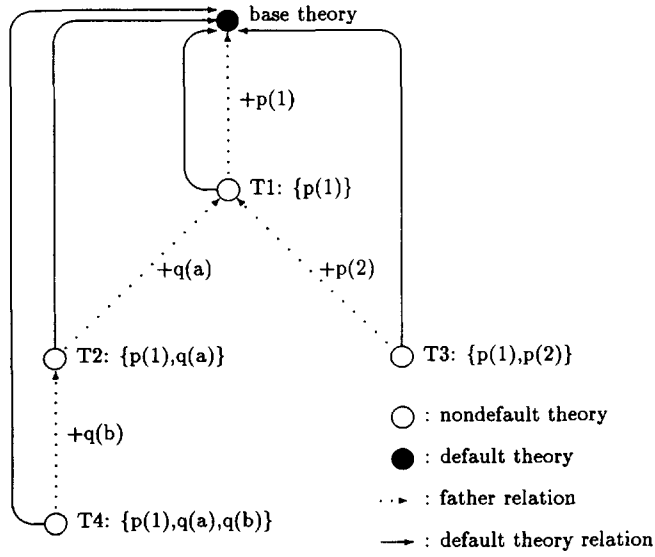


FIGURE 4. A theory tree in the nondefault theory approach.

default theory if it contains more procedures than a threshold number of procedures. In other cases, the system assumes that theory T is a nondefault theory. Of course, this decision can also be left to the user.

2.4. Context Switching

Since multiple theories are allowed in MetaProlog, we have to know at any time which context the system is in, and how to switch to another context whenever necessary. The MetaProlog system always runs in a certain context (also called the *current theory*), and all goals are proved with respect to this current theory. To prove a goal with respect to a certain theory T, first the context (current theory) is switched to T, and then the goal is proved with respect to the current theory.

In the MetaProlog system, there are two ways to switch context from one theory to another. The first one, called *temporary context switching*, is the context switching operation done by the predicate *demo*. The command *demo*(*Theory*, *Goal*) switches the context to *Theory*, and then *Goal* is proved with respect to the current theory. After the execution of this command, the context is automatically switched back to the previous context. The predicate *demo* can be defined in Prolog as follows:

```
demo(Theory, Goal):-
    context(PreviousContext),
    switch_context(Theory),
    call(Goal),
    switch_context(PreviousContext).
```

where the *context* predicate gets the current theory in which the system is currently running, and *switch_context* is a low-level system predicate that switches the context to the given theory.

The second one, called *permanent context switching*, is the context switching operation done by the *setcontext* predicate. The command *setcontext*(*TheoryName*), which is normally executed at the top level to define the context of the top level of the MetaProlog system, switches the context to the theory designated by *TheoryName*. After the execution of this command, the context is not switched back to the previous context. Of course, the context can be switched to another theory by submitting another *setcontext* command. The *setcontext* predicate can only be used with permanent theories (theories with names).

3. ABSTRACT METAPROLOG ENGINE

In 1983, Warren published a paper [41] describing an abstract machine for Prolog execution that consists of an abstract instruction set and several data areas on which the instructions operate. The model described in that paper for Prolog execution is now known as the Warren Abstract Machine (WAM). Many researchers in the logic programming community recognized the fact that the WAM represented a breakthrough in the design of Prolog systems and other computational logic systems. In fact, many commercial [2, 31] and noncommercial [9, 33] Prolog systems based on the WAM have been implemented after the introduction of the WAM. A full description of the WAM can be found in [1, 41]. After this point, we will assume that the reader is familiar with the WAM.

One of the main goals in our project was to achieve efficient implementation of MetaProlog. Since MetaProlog is an extension of Prolog, the best starting point was the WAM. For this purpose, the WAM was extended to an Abstract MetaProlog Engine (AMPE). Along the way, our own version of a WAM-based Prolog system [9] was created and then extended to the current MetaProlog system.

The AMPE can run in two different modes. The first one is the *simple* mode, in which the system runs when a two-argument *demo* predicate is encountered. The system runs in the *proof* mode when a three-argument or a four-argument *demo* predicate is encountered. The system can not only prove a goal with respect to a theory, but also collect the proof when it runs in the proof mode. Since the proof of a goal is collected when the system is in the proof mode, the system runs more slowly. However, the simple mode does not carry the burden of the proof mode. When it is needed, the system will switch from one mode to another during execution. The core part of the AMPE described in this section is used in both modes. But there are also extra features of the AMPE that are only used when it is in the proof mode. Proofs and their implementation are explained in Sections 4 and 5.

The core part of the AMPE is responsible for supporting multiple theories in the MetaProlog system. Since the MetaProlog system should be able to switch from one theory to another theory during execution, a fast context-switching mechanism is needed in the MetaProlog system. This task is accomplished by a theory register in the AMPE. This theory register is also saved in choice points, so that the context can be restored during backtracking.

Theories can be created on the fly during execution and discarded when the need for them disappears. So, the storage allocated for these theories should be reclaimed after they are discarded. In other words, their treatment should be similar to the treatment of structures and lists in the system. This observation suggests that the code area and the heap of the WAM should be integrated as a single data area in the AMPE.

The AMPE performs most of the functions of the WAM, but it also has some extra features to handle multiple theories of MetaProlog. These extra features of the AMPE in the simple mode are as follows:

1. A different memory organization, which is more suitable to handling compiled procedures and theories as data objects of the system.
2. Extra registers to handle theories in MetaProlog.
3. Functions of the procedural instructions in the AMPE that are different from their functions in the WAM.
4. A failure routine that should be able to switch to the proof mode during failure if it is necessary.

In the rest of this section, the core part of the AMPE will be discussed. In this discussion, the AMPE is widely compared with the WAM to explain similarities and differences between them.

3.1. Memory Organization of the AMPE

The memory of the AMPE is divided into three consecutive areas (Figure 5). The heap and the local stack grow from low memory to high memory, and the trail grows from high memory to low memory.

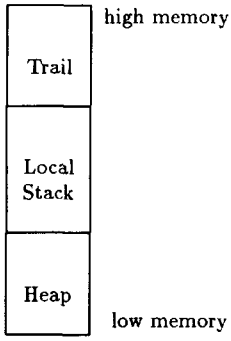


FIGURE 5. Memory organization of the AMPE.

The function of the local stack and the trail is the same as their function in the WAM, except that choice points carry extra information. Every choice point, whether it is created when the system is in the simple mode or in the proof mode, carries extra locations to store the current context (theory) and the current mode of the system. Choice points and environments that are created when the system is in the proof mode also carry extra information about proofs or branches. The implementation of proofs and branches is explained in Section 5.

The AMPE does not have a separate area in which to store code as the WAM does. Instead, the code area and the heap are integrated as a single data area in the AMPE. The heap holds compiled procedures and theory descriptors in addition to structures and lists. Compiled procedures and theory descriptors are represented by *boxes*, which are explained in Section 3.4.

Since the built-in predicates *addto* and *dropfrom* are backtrackable, the space held by the code created by these built-in predicates should be reclaimed during their failure. For example, the command *addto*($T1$, [$p(1)$, $p(2)$], $T2$) creates theory $T2$ by adding two clauses, namely $p(1)$ and $p(2)$, to theory $T1$. So it creates a new theory descriptor for $T2$, two compiled clauses, and an indexing block for the procedure $p/1$ on the heap. If backtracking occurs, theory $T2$ will be discarded, and all of the space used will be reclaimed if the space is not protected by another data structure in the heap. If the space is protected, it can be recovered during garbage collection. In other words, all unused space (held by theory descriptors, compiled procedures, or other data structures) in the heap can be reclaimed during backtracking or garbage collection.

3.2. Machine Registers

The AMPE has all the registers the WAM has, and it uses two extra registers to handle theories, and two registers to indicate the current mode of the AMPE. These are the only four new registers used when the system is in the simple mode. There are other extra registers used in the AMPE when the system is in the proof mode.

The registers that are the same as the WAM registers perform the same functions in the AMPE. For example, the program counter (P in the WAM) still points to the instruction to be executed, and the last choice point register (B in the WAM) still points to the last choice point in the local stack. Since the code area

and the heap in the WAM are integrated as a single data area in the AMPE, registers *P* and *CP* (program continuation pointer) point to this single data area in the AMPE.

The first new register is the *theory register TH*, which holds a pointer to the internal representation of a theory of the system. The register *TH* holds the current theory of the system, in which a procedure is searched for when a call to that procedure is encountered. The value of the *TH* is changed when the context of the system is switched to another context by the predicates *demo* or *setcontext*. The *theory register TH* is also saved in choice points, so that it can be restored from the value saved in the last choice point during backtracking.

The second register, the *theory counter register CTH*, is simply a counter that holds the next available theory-id, which is an integer. The function of the *theory counter register CTH* is to produce a unique theory-id for each theory in the system. When a new theory is created, this register is automatically incremented to hold the next available theory-id.

The *control register CTR* indicates the mode of the AMPE, and the *control information register CTRInfo* holds control information. When the system is in the simple mode, the register *CTR* contains flags indicating whether the system is in the simple mode, or it is in the proof mode and skipping the proof[2] of a goal. Information in registers *CTR* and *CTRInfo* is only used to decide whether the system has to switch to the proof mode or not during a failure when the system is in the simple mode. The function of these registers in the proof mode and failure routines in both modes are explained in detail in Section 5. These registers are also saved in choice points, so that the system can switch from one mode to another during backtracking.

3.3. Procedural Instructions

A MetaProlog program is directly compiled into instructions of the AMPE in the same manner as a Prolog program is compiled into instructions of the WAM. The instruction set of the AMPE is the same as the instruction set of the WAM, except that the functions of the procedural instructions differ from their functions in the WAM. Since each procedure in the WAM can be uniquely determined by its name and its arity, its address can be directly found when a *call* or an *execute* instruction is executed. On the other hand, when a *call* or an *execute* instruction is executed in the AMPE, the procedure is searched for in the current theory, which lives in the *theory register TH*. If the procedure is not found in the current theory, it is searched for in the default theory of the current theory, which is one of the ancestors of the current theory. A procedure is searched for among procedures of a theory using a hash function. This search continues recursively through default theories until the procedure is found, or backtracking occurs if it cannot be found. Figure 6 presents the search algorithm used to find a location of a procedure in instructions *call* and *execute*.

[2] Before the AMPE starts to skip the proof of a goal, it switches from the proof mode to the simple mode, and it runs in the simple mode until execution of that goal is completed. Then it switches back to the proof mode.

```
search_proc(Proc) {
/* move the current theory into a temporary register T */
T ← TH;
do {
    if Proc is found in T
        then {
            Loc ← location of Proc in T;
            return(Loc); }
        else
            T ← default theory of T;
} while {T is not the base theory}
/* Search is failed */
Fail; }
```

FIGURE 6. Theory search algorithm.

Although only the functions of the procedural instructions are different from their functions in the WAM when the system runs in the simple mode, the functions of some other instructions are also different from their functions in the WAM and in the simple mode of the AMPE when the system runs in the proof mode. In fact, the functions of the procedural instructions in the proof mode also differ from their functions in the simple mode. The functions of those instructions in the proof mode are presented in Section 5.

3.4. Data Types

Data types in the AMPE are similar to Warren's data types in the WAM, except that we have one extra data structure to hold untagged data, such as compiled clauses or theory descriptors. Untagged data in the AMPE, called *box*, are sealed between two tagged words.

Each object in the AMPE is represented by one or more 32-bit words. The low two, four, or eight bits of a word can be used as a tag. Two-bit tags are used to represent pointer data types such as references, structure (or box) addresses, and list addresses. An unbound variable is represented by a reference to itself. Four-bit tags are used to represent nonpointer one-word objects such as integers and functors. Eight-bit tags are used to represent objects consisting of more than one word; these are *boxes*. The low four bits of an eight-bit tag indicates that the object is a box, and the next four bits of the tag indicates the type of that box.

A *box* consists of consecutive words of memory such that the first and last words are box headers. Words between these box headers are untagged, and their formats depend on the type of box in question. Although the interior part of a box normally holds untagged words, it can also hold tagged words. Those tagged words should be located at word boundaries, and their positions in the box should be determined by the type of box. A *box header* is a word in the following format:

size of box	box type	box tag

The box tag shows that the word is a box header, and the box type shows the type of that box. The rest of the box header holds the size of that box in words. The format of the interior part of a box depends on the type of the box.

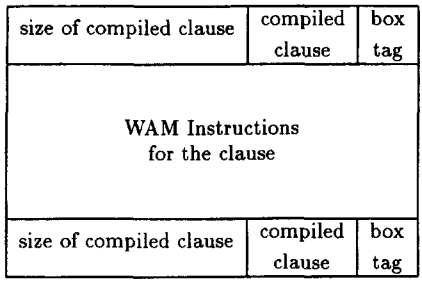


FIGURE 7. A compiled clause box.

For example, the box given in Figure 7 represents a compiled clause. The box headers at the beginning and at the end of the box show that it is a box for a compiled clause. The untagged part of that box contains WAM instructions for the clause, including indexing instructions such as *try_me_else*, *retry_me_else*, or *trust_me_else* as the first instruction of the clause.

Similarly, theory descriptors, index blocks, try-retry-trust blocks, and floating point numbers are represented by boxes. Of course, the formats of their untagged portions are different from each other. A theory descriptor contains a theory-id that uniquely identifies that theory, a pointer to its default theory, and pointers to compiled procedures belonging to the theory. A try-retry-trust block is a box whose untagged portion consists of a sequence of *try*, *retry*, and *trust* instructions. The untagged portion of an index block contains a *switch_on_term* instruction together with sequences of *try*, *retry*, and *trust* instructions.

The box header at the end of a box may appear unnecessary to the reader, but it plays an important role during garbage collection. It helps to identify the box when the heap is searched from top to bottom during garbage collection.

4. PROOFS

In MetaProlog, not only can goals be proved with respect to different theories, but their proofs can also be collected for future use. A proof is normally computed by execution of a three-argument *demo* predicate. A three-argument *demo* predicate represents a derivability relation between a theory and a goal with a certain proof. For example, if the command *demo*(*Theory*, *Goal*, *proof*(*Proof*)) is executed by the system, the variable *Proof* is bound to the proof of *Goal* in *Theory*.

Proofs are meta-level objects that have many applications in artificial intelligence, such as producing explanations in an expert system. For example, let *Carexpert* be a theory that represents an expert program written in MetaProlog to determine troubles in a given car. To find the problem in a given car, the following goal may be submitted:

$$demo(Carexpert, find\_trouble(Car, Problem)). \tag{4.1}$$

The variable *Car* is an input theory that contains information about a specific car or asks questions to get information about that car. The procedure *find_trouble* of the theory *Carexpert* finds the problem in a given car and returns *Problem* as an output. When the two-argument *demo* predicate in (4.1) is successfully executed, the trouble in the given car is found. The system can find the trouble in the given

car by using the two-argument *demo* predicate, but it cannot explain how it finds that trouble. To get the proof describing how *Carexpert* finds the trouble, the following goal should be submitted:

demo(*Carexpert*, *find_trouble*(*Car*, *Problem*), *proof*(*Proof*)). (4.2)

After the execution of this goal, the variable *Proof* will be bound to the proof of the predicate *find_trouble* in the theory *Carexpert*. The explanation for how *Carexpert* finds the trouble in the given car can be given later by examining *Proof*. We get this proof without any changes to the expert program *Carexpert*.

In the example above, *Proof* is a variable that is bound to the proof of the predicate *find_trouble* after the execution of the three-argument *demo* predicate in (4.2). However, *Proof* can also be partially instantiated to a proof before that goal is submitted. For example, assume that the procedure *find_trouble* can find out any kind of trouble in a given car. But we only want to find out troubles in its cooling system. To achieve this, we can instantiate *Proof* to a partial proof that forces the system to look for only trouble in the cooling system. In this case, we still do not need to change anything in the expert program *Carexpert*, but we can force the system to look for certain kinds of problems by giving a partial proof.

4.1. Structure of Proofs

The proof of a goal G in MetaProlog is a list whose head is an instance of G, and whose tail is a list of proofs of its subgoals. Of course, if it does not have any subgoals, its proof will be a singleton list. In Figure 8, patterns of proofs in two different cases are shown. In the first case, since the goal G is unified with a fact, the head of the proof of G is an instance of G, and the tail of proof is an empty list. In the second case, since G is unified with the head of a clause with one or more subgoals in its body, the tail of the proof of G is the list of subproofs of subgoals in that clause.

For example, let *Carexpert* be a theory containing the clauses given in Figure 9a. That theory represents a very simple expert program that finds the problem in a given car and suggests a solution to repair that problem. Clauses given in Figure 9b represent a problem in a specific car. To get a repair suggestion for the problem given in Figure 9b together with the proof of how that suggestion is found by the system, the following goal can be submitted:

demo(*Carexpert*, *repair_suggestion*(*Car*, *Suggestion*), *proof*(*Proof*)). (4.3)

After the execution of the three-argument *demo* predicate in (4.3), the variable *Suggestion* is bound to a term that represents a repair suggestion, and the variable *Proof* is bound to the following proof, which represents how the system gets that

Case 1	*Case 2*
Goal G: p(X)	Goal G: p(X).
Clause : p(a).	Clause : p(b) :- q(X), r(X).
Proof : [p(a)]	Proof : [p(b), <proof of q(X)>, <proof of r(X)>]

FIGURE 8. Structure of proofs.

```
repair_suggestion(Car,Suggestion) :-
    find_trouble(Car,Problem),
    get_suggestion(Problem, Suggestion).

get_suggestion(water_leak(Source), replace(Source)) :- hose(Source).
get_suggestion(water_leak(clamp), tighten(clamp)).
get_suggestion(oil_leak(oil_pan_bolt), replace(oil_pan_bolt)).

hose(radiator_hose).
hose(bypass_hose).

find_trouble(Car,Problem) :- check_cooling_system(Car,Problem).
find_trouble(Car,Problem) :- check_oil_system(Car,Problem).

check_cooling_system(Car,water_leak(Source)) :-
    demo(Car,leaking(water)),
    demo(Car,leaking_from(Source)).

check_oil_system(Car,oil_leak(Source)) :-
    demo(Car,leaking(oil)),
    demo(Car,leaking_from(Source)).
```

a. *Theory Carexpert*

```
leaking(water).
leaking_from(radiator_hose).
```

b. *Theory Car*

FIGURE 9. Theories for a simple expert system.

suggestion:

[*repair_suggestion*(⟨*theory car*⟩, *replace*(*radiator_hose*)),
[*find_trouble*(⟨*theory car*⟩, *water_leak*(*radiator_hose*)),
check_cooling_system(⟨*theory car*⟩, *water_leak*(*radiator_hose*)),
[*demo*(⟨*theory car*⟩, *leaking*(*water*)), [*leaking*(*water*)]]
[*demo*(⟨*theory car*⟩, *leaking_from*(*radiator_hose*)),
[*leaking_from*(*radiator_hose*)]]]]
[*get_suggestion*(*water_leak*(*radiator_hose*), *replace*(*radiator_hose*)),
[*hose*(*radiator_hose*)]]].

The head of the proof list above is an instance of our original goal in (4.3), and its tail is a list of proofs of subgoals of that goal. In the proof list above, ⟨*theory car*⟩ is a theory descriptor representing the theory *Car* in Figure 9b. After the proof above is collected by the system, an explanation can be given for why the system gets that repair suggestion by analyzing the collected proof. We can also submit the goal in (4.3) with a partial proof as follows:

demo(*Carexpert*, *repair_suggestion*(*Car*, *Suggestion*),
proof([*repair_suggestion*(*Car*, *Suggestion*),
[*find_trouble*(*Car*, *Problem*),
[*check_oil_system*(*Car*, *Problem*) | *SubProof*]]
| *RestofProof*])).

In this case, the third argument of the *demo* predicate is a partial proof that forces the system to look only for the trouble in the oil system of the given car.

After a successful execution of that goal, the partial proof is completed by the system. If there is no solution in the form given in the partial proof, the goal fails, even though there may be solutions in some different form.

4.2. *Skipping Proofs*

When a three-argument *demo* predicate, *demo*($T, G,$ *proof*(P)), is successfully executed, the variable P is bound to a proof of G in T. This proof contains the proofs of all subgoals of G. Although proofs are useful in many applications, all details of proofs may be unnecessary in some cases. We should not pay the extra cost to collect these unnecessary parts of proofs in those cases.

In the MetaProlog system, certain subproofs of a proof can be skipped by using a four-argument *demo* predicate instead of a three-argument *demo* predicate. The fourth argument of this *demo* predicate contains control information about subgoals of the goal given in that *demo* predicate. This control information is a list of procedures whose proofs are skipped during the execution of the given goal. Continuing with our *Carexpert* example in the previous subsection, let us assume that we are only interested in how the system gets a repair suggestion for a problem, but we do not care how it finds that trouble in a given car. In other words, we do not care about the proof of the subgoal, *find_trouble*(*Car*, *Problem*). To skip the proof of that subgoal, the following four-argument *demo* can be submitted:

demo(*Carexpert*, *repair_suggestion*(*Car*, *Suggestion*),
proof(*Proof*), *skip*([*find*_trouble/2])).

During the computation of the goal above, the proof of the procedure *find_trouble* is skipped (i.e., its proof is not collected), and the proof of the goal is bound to the following term:

[*repair_suggestion*(⟨*theory car*⟩, *replace*(*radiator_hose*)),
[*find_trouble*(⟨*theory car*⟩, *water_lead*(*radiator_hose*)) | ⟨*skipped proof*⟩],
[*get_suggestion*(*water_leak*(*radiator_hose*), *replace*(*radiator_hose*)),
[*hose*(*radiator_hose*)]]].

In the proof term above, ⟨*skipped proof*⟩ is a constant that represents a skipped proof.

4.3. *Fail Branches*

In the previous sections, only proofs that are just success branches in a search tree are discussed. In this section, fail branches of a search tree and how they are collected in the MetaProlog system are discussed.

When the following three-argument *demo predicate* is executed in the MetaProlog system, *Branch* is bound to the leftmost branch of the search tree of *Goal* relative to *Theory*:

demo(*Theory*, *Goal*, *branch*(*Branch*)).

Backtracking into this *demo* predicate will cause *Branch* to be bound to the successive branches of the search tree. This branch can be a success branch (proof) or a fail branch of the search tree.

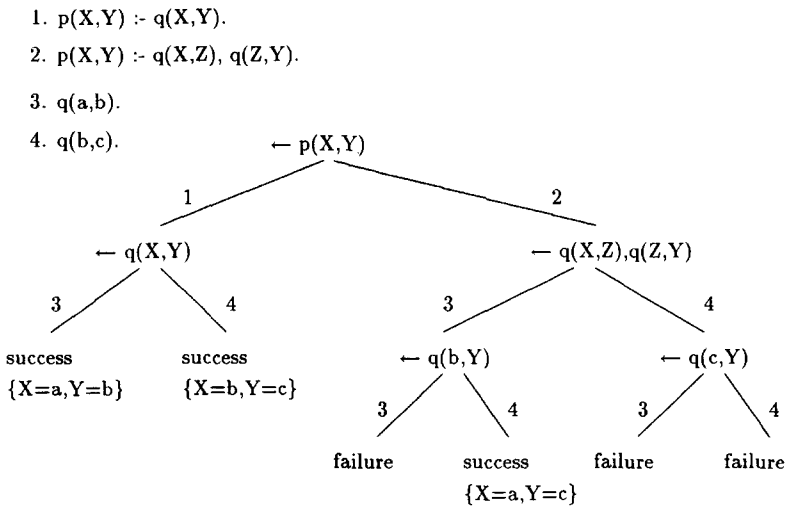


FIGURE 10. A trivial theory and its search tree.

In Figure 10, a trivial theory T and a search tree of the goal $p(X,Y)$ relative to theory T are given. In the search tree, there are three success branches and three fail branches. After the execution of *demo*(T, $p(X,Y)$, *branch*(*Branch*)), the variable *Branch* is bound to the leftmost branch of the search tree. The branches in Figure 10 are represented in the MetaProlog system as follows:

1st Branch: `[p(a, b), [q(a, b)]]`
2nd Branch: `[p(b, c), [q(b, c)]]`
3rd Branch: `[p(a, Y), [q(a, b)], [q(b, Y), fail]]`
4th Branch: `[p(a, c), [q(a, b)], [q(b, c)]]`
5th Branch: `[p(b, Y), [q(b, c)], [q(c, Y), fail]]`
6th Branch: `[p(b, Y), [q(b, c)], [q(c, Y), fail]]`.

Each fail branch has exactly one atomic fail subbranch. An atomic subbranch is a list whose head is a subgoal and whose tail is the list [*fail*]. For example, the atomic fail branch of the third Branch above is the following term:

```
[q(b, Y), fail].
```

An atomic fail branch separates a fail branch into two parts. The first part is the collected part of the fail branch, and the second part is the uncollected part of the fail branch. Even though fail branches are not completely collected, their collected parts are enough to give the reason for that failure. The collected part will reflect all unifications occurring before the failure, and the atomic fail subbranch will reflect the exact location of that failure.

5. PROOF MACHINE

The system normally switches from the simple mode to the proof mode when it encounters a three-argument or four-argument *demo* predicate when it is running in the simple mode. Of course, if the system encounters those predicates in the

proof mode, it stays in the proof mode. In the proof mode, a success or fail branch of a goal is also collected as it is developed by the underlying theorem prover. The system can also be forced to collect certain specified branches of the search tree of a goal by giving a partially instantiated branch in the three-argument or four-argument *demo* predicate.

In the proof mode of the AMPE, extra mechanisms are used to support proofs in MetaProlog, in addition to the mechanisms used in the simple mode for the implementation of multiple theories and context switching among these theories. The simple mode of the AMPE may be called the *simple machine*, and the proof mode of the AMPE the *proof machine* after this point. The proof machine assigns different meanings to the procedural instructions, and it uses extra registers to handle proofs in addition to the basic mechanism used in the simple mode of the AMPE. In the rest of this section, properties of the proof machine are discussed.

5.1. Registers in the Proof Mode

The proof machine uses two new extra registers to collect proofs. The first one, the *proof register Pr*, points to the part of the proof that is currently being collected by the system. The second one, the *continuation proof register CPr*, points to the part of the proof that will be collected by the system after the part of the proof indicated by the proof register *Pr* is collected. There is a very close analogy between the proof register *Pr* and the program pointer register *P*, as well as between the proof continuation register *CPr* and the continuation program pointer register *CP*.

These two new registers are initialized with parts of a proof template for oncoming proof computation when a three-argument or four-argument *demo* predicate is executed. The values pointed to by these registers are unified with parts of proofs when the procedural instructions are executed in the proof mode. Section 5.2 explains how they are exactly handled by procedural instructions.

The continuation proof register *CPr* is also saved in environments by the *allocate* instruction, and restored by the *deallocate* instruction in the proof mode. Environments created in the proof mode are marked by a bit to show that they were created in the proof mode. The existence of this mark bit does not lead to any overhead in the simple mode, because this mark bit is only used in the proof mode or during garbage collection [13].

Choice points created in the proof mode differ from choice points created in the simple mode. Choice points created in the proof mode hold values of the registers *Pr* and *CPr* in addition to other values that are saved in choice points created in the simple mode. The type of a choice point can be determined by the saved value of the control register *CTR* in that choice point.

The control register *CTR* is a flag register that holds certain flags to determine different situations in the AMPE. Two groups of flags can be used in the register *CTR* when the system is in the proof mode. There are two flags in the first group, and they are set when a three-argument or four-argument *demo* predicate is executed. Values of these flags are not changed during the execution of that *demo* predicate unless another *demo* predicate is executed as a subgoal. The first flag is the *mode flag* used to determine the mode of the system. In this case, that flag will be set to a value to indicate that the system is in the proof mode. The second flag is

the *branch flag*. The branch flag is 0 when the system collects only success branches (proofs) of a goal, and it is 1 when the system collects all branches of a goal, including fail branches. The second group contains *skip flags*, which are used when proofs of certain procedures are skipped.

The content of the control information register *CTRInfo* can be either the constant "nil" or a pointer to a control information box that holds certain control information in the proof mode. It will point to a control information box when proofs of some procedures are skipped, or all branches of the search tree of a goal are collected.

A control information box has three slots. The first slot is a pointer to a list of procedures whose proofs will be skipped. The second and third slots hold an environment pointer and a program pointer, respectively. If a failure occurs when the system collects all branches, the environment pointer register E and the program pointer register P are restored from values stored in those slots, and execution continues from the location stored in the third slot. Of course, before control is transferred to that location, a fail branch is collected.

5.2. *Machine Instructions in the Proof Mode*

The functions of some indexing instructions and all procedural instructions in the proof mode are different from their functions in the simple mode. These instructions have to do extra work to collect proofs or to save extra information in choice points. The indexing instructions *try_me* and *try* instructions save two new registers *Pr* and *CPr* in choice points, in addition to values saved in choice points created in the simple mode when they are executed in the proof mode. Procedural instructions are also responsible for collecting proofs of procedures in addition to transferring control to those procedures.

When a *call* instruction for a procedure p/n is executed, the content of the proof register *Pr* is unified with a list in the following form:

$$[[p(A_1,\ldots,A_n) \mid \mathit{SubProofs}] \mid \mathit{RestofProof}]. \tag{5.1}$$

The term above represents a part of the proof of the calling procedure. The head of the list above represents a proof of the procedure p/n, and the tail of that list, *RestoProof*, represents proofs of later subgoals in the procedure, which calls p/n. The head of the proof list, $p(A_1,\ldots,A_n)$, is the term whose functor is p and whose ith argument is a value unified with the ith argument register. The tail of the proof list, *SubProofs*, represents proofs of subgoals in the body of a clause of the procedure p/n. The algorithm of *call* instruction in the proof mode is given in Figure 11a. If any unification in the algorithm given in Figure 11a fails, backtracking occurs. This can only happen if a partial (or complete) proof is passed in a *demo* predicate to choose only certain branches in a search tree. Before control is transferred to the procedure p/n, the proof register *Pr* is set to point to the tail of the proof of the procedure p/n, *SubProofs* in (5.1), to compute the proof of the procedure p/n. Since the procedure p/n is not the last subgoal in the calling procedure, we have to continue to collect the rest of this calling procedure after the proof of the procedure p/n is collected. For that reason, the register *CPr* is set to point to that part of the proof, *RestoProof* in (5.1), of the calling procedure.

```
proofpart ← createlist;                 % Create a list structure on the heap
unify(Pr,proofpart);                    % Unify it with the content of Pr
prooflist ← createlist;                 % Create a list structure on the heap
unify(head(proofpart),prooflist);       % Unify it with the head of "proofpart"
proofhead ← createterm(p,n);            % Create term p/n
unify(head(prooflist),proofhead);       % Unify it with the head of "prooflist"
for(i = 1; i ≤ n; i ← i + 1)            % Unify its i^th argument with A_i
   unify(A_i,argument(proofhead,i));
Pr ← tail(prooflist);                   % Set Pr to point to the tail of "prooflist"
CPr ← tail(proofpart);                  % Set CPr to point to the tail of "proofpart"
CP ← nextlocation(P);                   % Set CP to point to the next instruction
P ← location(p/n);                      % Jump to procedure p/n in current theory
```

a. *Call Instruction*

```
proofpart ← createlist;                 % Create a list structure on the heap
unify(Pr,proofpart);                    % Unify it with the content of Pr
unify(tail(proofpart),[ ]);             % Unify its tail with the symbol "nil"
prooflist ← createlist;                 % Create a list structure on the heap
unify(head(proofpart),prooflist);       % Unify it with the head of "proofpart"
proofhead ← createterm(p,n);            % Create term p/n
unify(head(prooflist),proofhead);       % Unify it with the head of "prooflist"
for(i = 1; i ≤ n; i ← i + 1)            % Unify its i^th argument with A_i
   unify(A_i,argument(proofhead,i));
Pr ← tail(prooflist);                   % Set Pr to point to the tail of "prooflist"
P ← location(p/n);                      % Jump to procedure p/n in current theory
```

b. *Execute Instruction*

```
unify(Pr,[ ]);                          % Unify the content of Pr with an empty list
Pr ← CPr;                               % Move the content of CPr into Pr
P ← CP;                                 % Move the content of CP into P
```

c. *Proceed Instruction*

FIGURE 11. Algorithms of procedural instructions in proof mode.

On the other hand, when an *execute* instruction for the procedure p/n is executed, the content of the proof register *Pr* is unified with a singleton list, whose only element is the proof list of that procedure. The term unified with the content of the register *Pr* is as follows:

$$[[\,p(A_1,\ldots,A_n) \mid \mathit{SubProofs}]]. \tag{5.2}$$

Since the procedure p/n is the last subgoal in the calling procedure, the *execute* instruction is responsible for completing the proof of the calling procedure with the term above. The full algorithm of the *execute* instruction is given in Figure 11b. If all unifications in the algorithm given in Figure 11b are successful, the register *Pr* is set to point to the tail of the proof of the procedure p/n, *SubProofs* in (5.2), for the oncoming proof computation of that procedure. Since the procedure p/n is the last subgoal in the calling procedure, we do not need to update the register *CPr*.

When a *proceed* instruction is executed in the proof mode, the content of the proof register *Pr* is unified with an empty list, and the content of the continuation proof register *CPr* is moved to the register *Pr*. The algorithm of the *proceed* instruction is given in Figure 11c.

```
p(X,Y) :-          p/2:              q/1:
   q(X), r(Y).       allocate 1        get args of q/1
                     get args of p/2   proceed
q(a).              put args of q/1
                   call q/1,1
r(b).              put args of r/1   r/1:
                   deallocate          get args of r/1
                   execute r/1         proceed
```

FIGURE 12. A trivial program and its WAM instructions.

Example 5.1. In this example, we will use a trivial program to represent a theory T, and we will trace the proof computation of a goal in that theory. In this trace, we show how registers *Pr* and *CPr* are changed by procedural instructions, and which part of the proof is collected at each step. Now let us assume that the trivial program in Figure 12 represents the theory T. The first clause of that trivial program has two subgoals, and the second and third clauses are unit clauses. Figure 12 also gives WAM instructions of that trivial program after its compilation. The compiled clauses of the procedures $q/1$ and $r/1$ have a *proceed* instruction, since they are unit clauses. The compiled clause of the procedure $p/2$ has a *call* instruction and an *execute* instruction. When the predicate *demo*(T, $p(X,Y)$, *proof*(*Proof*)) is executed, the contents of registers *Pr* and *CPr* and the proof being collected are changed as shown in the trace given in Figure 13. Before control is transferred to the procedure $p/2$, a proof template list whose head is the goal in the *demo* predicate and whose tail is an unbound variable is created, and the register *Pr* is set to point to the tail of this list. Later, each procedural instruction adds new proofs to this list and updates registers *Pr* and *CPr* properly, depending on that procedural instruction.

5.3. *Implementation of the demo Predicate*

A branch of a search tree of a goal can be collected by a three-argument or four-argument *demo* predicate. A branch collected by a *demo* predicate can be a fail branch or a success branch (proof), depending on the arguments of that *demo*

```
At the beginning                Proof: [ p(X,Y) | _ ]
of procedure p/2                              ↑Pr

After the execution             Proof: [ p(X,Y), [ q(X) | _ ] | _ ]
of call q/1 in p/2                           ↑Pr  ↑CPr

After the execution             Proof: [ p(a,Y), [ q(a) ] | _ ]
of proceed in q/1                            ↑Pr

After the execution             Proof: [ p(a,Y), [ q(a) ], [ r(Y) | _ ] ]
of execute r/1 in p/2                                  ↑Pr

After the execution             Proof: [ p(a,b), [ q(a) ], [ r(b) ] ]
of proceed in r/1
```

FIGURE 13. Trace of execution of a trivial program.

predicate. If the third argument of a *demo* predicate is *proof*(*P*), only success branches will be collected. On the other hand, if it is *branch*(*B*), all branches, including fail branches, of the search tree of the given goal are collected by the system.

The fourth argument of a *demo* predicate gives control information to be used while proving that goal. The control information is a list of procedures whose proofs will be skipped during execution of a goal in that *demo* predicate. The third argument of a *demo* predicate also carries a kind of control information. It determines whether only success branches or all branches are going to be collected.

A three-argument *demo* predicate is the same as a four-argument *demo* without a skip list. In the MetaProlog system, a four-argument *demo* predicate is defined as follows:

```
demo(Theory, Goal, Branch, SkipPreds):-
  context(CurrTheory),
  proof_context(CurrCTR, CurrCTRInfo, CurrPr, CurrCPr),
  switch_context(Theory),
  switch_proof_mode(Branch, SkipPreds),
  call(Goal),
  reset_proof_mode(CurrCTR, CurrCTRInfo, CurrPr, CurrCPr),
  switch_context(CurrTheory).
```

The predicates *context* and *proof_context* get the current values of the registers *TH*, *CTR*, *CTRInfo*, *Pr*, and *CPr* to be restored after the execution of *Goal* is completed. The predicate *switch_context* sets the current theory register *TH* to the given theory in the *demo* predicate. The predicate *switch_proof_mode* prepares the registers *CTR*, *CTRInfo*, *Pr*, and *CPr* for oncoming proof computation. These registers are set as follows, depending on values of the third and fourth arguments of the *demo* predicate. After these registers are set, execution continues in the proof mode.

CTR

- Mark the register *CTR* to indicate that the AMPE will be in the proof mode.
- Mark the register *CTR* to indicate that all branches of the search tree will be collected if the third argument is in the form of *branch*(*B*).

CTRInfo

- Create a new control information box with three slots if there is any procedure whose proof will be skipped or if all branches will be collected. If a control information box is created, we continue to fill this control information box; otherwise, the register *CTRInfo* is set to constant "nil."
- Set the first slot of the control information box to the list of procedures whose proofs will be skipped if there is any; otherwise, set it to constant "nil."
- If all branches are to be collected, save environment pointer *E* in the second slot, and a pointer to the body of the *demo* predicate in the third slot. This code pointer points to the location of subgoal *reset_proof_mode* in the body of the *demo* predicate, and the saved environment pointer points to the

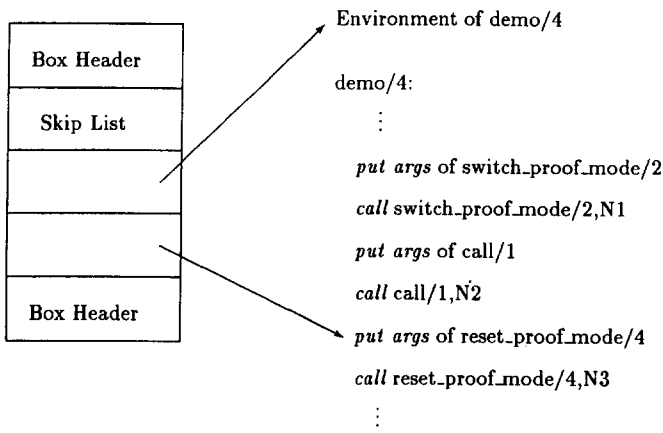


FIGURE 14. A control information box and *demo* predicate.

environment of the four-argument *demo* predicate (cf. Figure 14). These values are used to transfer control to the four-argument *demo* during a failure.

Pr and **CPr**

- Create a proof template on the heap for oncoming proof computation, set register *Pr* to the head of that proof template, and register *CPr* to the tail of that template. This proof template is as follows:

  $$[[call(Goal), Branch] \mid Rest].$$

  This template represents a part of the proof of the four-argument *demo* predicate. The head of this template represents a proof of goal *call*(*Goal*). The proof of *call*(*Goal*) is a singleton list whose element is the proof of *Goal*.

After the execution of *call*(*Goal*) is completed, *Branch* is bound to the leftmost branch of the search tree of *Goal* relative to *Theory*. The predicates *reset_proof_mode* and *switch_context* restore old values of the registers *CTR*, *CTRInfo*, *Pr*, *CPr*, and *TH*, and execution continues in the simple mode or in the proof mode, depending on the restored value of the register *CTR*.

5.4. *Implementation of Skipping of Branches*

When a *call* or *execute* instruction is executed for a procedure whose proof should be skipped, the system switches from the proof mode to the simple mode to skip its proof. The algorithms for *call* and *execute* instructions in Section 5.2 should check whether the proof of the procedure will be skipped. If it will be skipped, they have to switch the mode of the AMPE from the proof mode to the simple mode before they transfer control to that procedure.

Before control is transferred to a procedure whose proof will be skipped, the proof register *Pr* points to the tail of the proof of that procedure. Since a singleton list is chosen to represent a skipped proof in the MetaProlog system, the content of

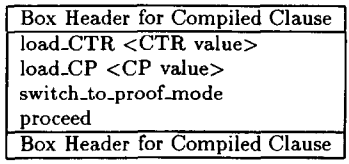


FIGURE 15. A skip break point.

the proof register *Pr* should be unified with an empty list after the execution of the procedure is completed in the simple mode. In other words, the proof of that procedure will be a singleton list whose only element is an instance of that procedure call.

After the execution of that procedure is completed in the simple mode, the mode of the AMPE has to be switched back to the proof mode to continue to collect the proof of the calling procedure. But the system cannot know when the execution of that procedure is completed in the simple mode. This problem is solved by creating a skip break point that will be executed after the procedure is completed in the simple mode. Normally, when the execution of a procedure is completed, the execution continues from the location pointed at by the continuation program pointer register *CP*. This is the location at which execution must be interrupted at the end of the procedure whose proof is skipped in the simple mode. Before control is transferred to that procedure, a skip break point (cf. Figure 15) is created on the heap. The register *CP* is set to point to the beginning of instructions of this skip break point, and the register *CTR* is marked to indicate that the system will be skipping a branch of the search tree.

A skip break point is a compiled clause that contains four AMPE instructions. Three of these four instructions are new AMPE instructions, and they are executed in the simple mode. The last one is a *proceed* instruction. When a skip break point is created, the current values of registers *CTR* and *CP* are saved in the first two instructions of this skip break point. These first two instructions, *load_CTR* and *load_CP*, restore the original values of these registers. When the third instruction, *switch_to_proof_mode*, is executed, the system switches back to the proof mode. When the last instruction, *proceed*, in the skip break point is executed in the proof mode, the content of the proof register is bound to an empty list, and execution continues from the location indicated by the register *CP*.

5.5. Failure Routines

When a failure occurs, an earlier state of the computation should be restored from the values stored in the last choice point. Since the mode of the system is a part of a state of computation in the AMPE, failure routines should be able to switch from one mode to the other. They should also be able to collect a fail branch when the system collects all branches in a search tree.

First, failure routines in both modes check whether the system is collecting all branches of a search tree. If the system is collecting all branches, a fail branch of that search tree is collected by failure routines. Since we collect a failure branch of a subgoal whose proof is skipped when there is no success branch in the search tree of that subgoal, the failure routine in the simple mode checks this condition before it collects that fail branch. The following actions are taken when the

conditions above are satisfied in either mode:

- The content of the proof register *Pr* is unified with the term [*fail*], which represents a part of a fail branch.
- Since the execution should continue in the last *demo* predicate that caused the collection of all branches in a search tree, the environment of this *demo* predicate is restored from the control information box in the register *CTRInfo*. Then control is transferred to the location stored in that control information box after the mode of the system is switched back to the proof mode.

If all branches of the search tree are not being collected, failure routines in both modes perform functions similar to those of the failure routine in the WAM. First, trailed entries are untrailed, and registers are restored from values stored in the last choice point. The registers *Pr* and *CPr* are restored if that choice point is created in the proof mode; otherwise, they are not restored. The system switches to the mode determined by the restored register *CTR*.

6. RELATED WORK

Many proposals to extend logic programming with theories (modules or contexts) and proofs have been presented [12, 21–27, 30] in the literature. Some of them are just interested in the expressive power, flexibility, and declarative semantics of meta-level facilities without emphasizing implementation techniques for these meta-level facilities. Since our paper is mainly about the implementation of the meta-level facilities in MetaProlog, and since the work done by Lamma et al. [22, 24] and the work done by Nadathur et al. [30] are the only studies that concentrate on the implementation of contexts, we will compare our work with theirs. There are two main differences between our system and the other two systems mentioned above:

- In those systems, contexts are implicitly created to be used only in the derivation of a single goal, whereas we have explicit handles (MetaProlog variables bound to theory descriptors) to contexts, so that they can be used in derivations of more than one goal. This difference can explain the reason for the usage of a stack for the implementation of contexts in those systems, and the usage of the heap to keep theories in our system.
- To locate a procedure in a context may be faster in the MetaProlog system than in those systems, because of the nondefault theory approach in our system.

In the rest of this section, these differences are discussed in detail and justifications for some of our design decisions are given.

The system developed by Lamma et al. [22–24] for contextual logic programming [27] has units that are sets of clauses identified by Prolog constants, and contexts that are ordered lists of units. The contexts in their system are closely related to the theories in our system. In their system, when an extended goal $u \gg G$ is executed, a new context is created from the current context by adding clauses of the unit u to the current context before starting the derivation of the goal G. This new context is only used in the derivation of the goal G, and it is only

implicitly accessible during its derivation. If the goal G is deterministic, the new context is automatically discarded after the completion of its derivation. If it is nondeterministic, the new context is not discarded, so that backtracking to that goal can be possible. Since there is no explicit handle for a context in their system, a variable cannot be bound to it, and it cannot be returned as a value. At the implementation level, the unit u is pushed into a stack to create a new context when the extended goal $u \gg G$ is encountered. If the goal G is deterministic, the pushed unit is popped from the stack to discard that new context after the derivation of G. The new stack that is used to implement contexts in their system is called the context stack.

In the system developed by Nadathur et al. [30], a context is also created for the derivation of a goal and discarded afterward. The addition of clauses takes place as a result of an implication goal. The clauses in an implication goal are added to the program before solving the goal in that implication goal. If the goal is deterministic, the need for the context created for this goal disappears, and the space occupied by that context can be reclaimed in their system. Again, a stack-based mechanism is appropriate for the implementation of contexts in their system, as in the system developed by Lamma et al. They push a special environment (called an implication record) into the local stack to create a new context and pop this implication record to discard this context. The main difference between these two systems is that the local stack is used to keep contexts in the system of Nadathur et al. instead of a separate context stack, which is used in the system developed by Lamma et al.

On the other hand, a theory is explicitly created by the predicates *addto* and *dropfrom* in the MetaProlog system, and an explicit handle to that theory is returned as a value. After the creation of a theory, the variable given in the third-argument position is bound to the internal structure of this new theory. A theory is accessible as long as the variable bound to its internal representation is accessible. A theory can be used in derivations of more than one goal once it is created. The life cycle of a theory depends on the life cycle of the variable bound to that theory in the MetaProlog system. The life cycle of a theory is similar to the life cycle of a structure in the heap. In fact, this is the reason why we keep theories in the heap by removing the code area in our WAM-based system. We cannot put theories into a stack, because their life cycles are not restricted by the life cycle of the derivation of a single goal. For example, the MetaProlog goal

addto(*OldTheory*, *Clauses*, *NewTheory*), *demo*(*NewTheory*, *Goal*)

can be simulated by

*unit*1 $\gg$ *Goal*

in the contextual logic programming (CxLP), if the unit *unit*1 contains the same clauses in *Clauses* and the current context contains the same clauses in *OldTheory*. In the system developed by Lamma et al., a new context, which will be equal to *NewTheory*, is created by adding clauses in the unit *unit*1 to the current context, and it is only used during the derivation of *Goal*. On the other hand, in the MetaProlog system, the new theory, to which *NewTheory* is bound, will not be automatically discarded after the execution of the *demo* predicate, and the same

theory can be returned as a value to be used in other *demo* predicates to prove different goals with respect to that theory, or in theory creation predicates such as *addto* to create another new theory from that theory.

In the system developed by Lamma et al., they adopt an explicit representation of the context as a set of units. The code of each unit is composed of procedures explicitly defined in that unit. To access a procedure in the current context, that context is searched in the units of the current context. For example, if the current context is the ordered set of units $[U_n, U_{n-1}, \ldots, U_1]$, and we want to access the procedure p/n in the current context, that procedure is searched in the unit U_n. If it is not found in U_n, then it is searched in U_{n-1}. This search continues until the procedure p/n is found or no more units are left to be searched. In their system, the cost of accessing a procedure will depend on the number of units in the current context. The search can be expensive if there are a lot of recursive extended goal invocations. In this discussion, we assumed that the call to the procedure p/n is a lazy goal of their system. In their system, the right code for local and eager goals is found at compile or extension times, respectively.

In the system developed by Nadathur et al., the time needed to locate the code of a predicate in a context is proportional to the number of the nesting levels of implication goal invocations in that context. They have to search the right code for a predicate in implication records created on the local stack for each implication goal. The search starts from the most current implication record on the top of the stack and continues until the predicate is found or the bottom of the stack is reached. This can be expensive if there are a lot of recursive implication goal invocations.

In our system, to access a procedure in a theory T, that procedure is searched in T. If it is not found there, it is searched in the default theory of T. This search proceeds with default theories along the branch from T to the base theory. The procedure is searched in each theory using a hash function. The maximum number of theories that may be searched is the number of default theories on the branch from T to the base theory. In fact, this is the reason why the nondefault theory approach is used in the representation of theories instead of the parent relation among theories. In our system, most theories will be nondefault theories, and a few of them will be default theories. Thus the number of default theories that will be searched to access a procedure will be small, and the search for that procedure will not be expensive. If we make all theories in our system default theories, a search for a predicate will be similar to the search in the system developed by Lamma et al.

Since we are not aware of any WAM-based system that collects proofs and uses them to shrink search spaces of goals in the literature, we are not able to compare our implementation with any other system. There are meta-interpreters that collect proofs [36–40], but they do not efficiently implement proofs, because of the extra layer of interpretation. To the best of our knowledge, our system is the only system in the literature that deals with proofs at the WAM level. To handle proofs, we use extra registers and assign different meanings to procedural instructions. These extensions are smoothly integrated with the original WAM.

7. CONCLUSION

To implement meta-level facilities such as theories, proofs, fail branches, and control knowledge in a WAM-based system, we have to extend the WAM by

redefining meanings of indexing and procedural instructions, introducing new registers to be used in proof computations, and loading extra tasks to failure routine. To be able to explicitly represent meta-level objects and control knowledge, some guidelines for changes to the WAM can be summarized as follows:

1. New registers may be needed to get efficient implementation of these meta-level objects. We had to introduce the theory register *TH* to handle multiple theories, and the proof register *Pr* and the proof continuation register *CPr* to handle proofs.
2. Explicit representation of control knowledge may be handled by storing this control knowledge in some data structure pointed at by some new registers. This task is handled by *CTR* and *CTRInfo* registers in the MetaProlog system. To achieve more efficiency, some portions of control knowledge may be implemented by using separate mechanisms, but they may increase the complexity of the architecture.
3. Environments and choice points may need to keep extra information about meta-level objects and control knowledge.
4. Most of the mechanisms introduced to handle meta-level objects and control knowledge are used by procedural instructions and failure routines.

If the guidelines above are followed, new control knowledge may easily be added to a WAM-based system. For example, to control the depth of a proof tree, extra information can be included in control information boxes, and that extra information can be used by procedural instructions. To do this, control information should keep a new value indicating the maximum depth of a proof tree, and a new global counter should hold the current depth. Procedural instructions may check whether the maximum depth is reached by comparing those two values.

Thanks to Ken Bowen for valuable feedback during this research. This paper has benefitted from the suggestions for improvements by its anonymous reviewers.

REFERENCES

1. Aït-Kaci, H., *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, Cambridge, MA, 1991.
2. *ALS Prolog Reference Manual*, Applied Logic Systems, 1988.
3. Attardi, G. and Simi, M., Metalanguage and Reasoning across Viewpoints, in: *Proc. 6th ECAI*, Pisa, Italy, 1984.
4. Bacha, H., Meta-Level Programming: A Compiled Approach, in: *Proc. 4th Int. Conf. Logic Programming*, MIT Press, Cambridge, MA, 1987, pp. 394–410.
5. Bacha, H., MetaProlog Design and Implementation, in: *Proc. 5th Int. Conf. Symp. Logic Programming*, MIT Press, Cambridge, MA, 1988, pp. 1371–1387.
6. Bowen, K. A. and Kowalski, R. A., Amalgamating Language and Metalanguage in Logic Programming, in: K. Clark and S.-A. Tarnlund (eds.), *Logic Programming*, Academic Press, London, 1982, pp. 153–173.
7. Bowen, K. A. and Weinberg, W., A Meta-Level Extension of Prolog, in: *Proc. 1985 Symp. Logic Programming*, IEEE Computer Society Press, Washington, DC, 1985, pp. 48–53.
8. Bowen, K. A., A Meta-Level Programming and Knowledge Representation, *New Generation Comput.* 3:359–383 (1985).

9. Bowen, K. A., Buettner, K. A., Cicekli, I., and Turk, A., A Fast Incremental Portable Prolog Compiler, *Lecture Notes Comput. Sci.* 225:650–656 (1986).
10. Bratko, I., *PROLOG Programming for Artificial Intelligence*, 2nd edition, Addison-Wesley, New York, 1990.
11. Bruffaerts, A. and Henin, E., Negation as Failure: Proofs, Inference Rules and Meta-Interpreters, in: H. Abramson and M. H. Rogers (eds.), *Meta-Programming in Logic Programming*, MIT Press, Cambridge, MA, 1989, pp. 169–190.
12. Bugliesi, M., Lamma, E., and Mello, P., Modularity in Logic Programming, *J. Logic Programming* 19/20:443–502 (1994).
13. Cicekli, I., A Garbage Collector for the MetaProlog System (or: Collecting All the Garbage in Prolog Systems), Logic Programming Research Group Technical Report LPRG-TR-88-2, Syracuse, NY, 1988.
14. Cicekli, I., Design and Implementation of an Abstract MetaProlog Engine for MetaProlog, in: H. Abramson and M. H. Rogers (eds.), *Meta-Programming in Logic Programming*, MIT Press, Cambridge, MA, 1989, pp. 417–434.
15. Cicekli, I., Design and Implementation of an Abstract MetaProlog Engine for MetaProlog, Ph.D. Dissertation, Syracuse University, Syracuse, NY, 1991.
16. Clocksin, W. F. and Mellish, C. S., *Programming in Prolog*, 2nd edition, Springer Verlag, New York, 1984.
17. Dincbas, M. and Le Pape, J., Metacontrol of Logic Programs in METALOG, in: *Proc. Int. Conf. Fifth Generation Comput. Syst.*, Tokyo, 1984.
18. Eshghi, K., Applications of Meta-Language Programming to Fault Finding in Logic Circuits, in: *Proc. 1st Int. Conf. Logic Programming*, Marseille, 1982.
19. Eshghi, K., *MetaLanguage in Logic Programming*, Ph.D. Dissertation, Imperial College, London, 1986.
20. Gallaire, H. and Lasserre, C., A Control Metalanguage for Logic Programming, in: *Proc. Logic Programming Workshop*, 1980.
21. Giordano, L. and Martelli, A., A Modal Reconstruction of Blocks and Modules in Logic Programming, in: *Proc. 1991 Int. Symp. Logic Programming*, MIT Press, Cambridge, MA, 1991, pp. 239–253.
22. Lamma, E., Mello, P., and Natali, A., The Design of an Abstract Machine for Efficient Implementation of Contexts in Logic Programming, in: *Proc. 6th Int. Conf. Logic Programming*, MIT Press, Cambridge, MA, 1989, pp. 303–317.
23. Lamma, E., Mello, P., and Natali, A., Reflection Mechanisms for Combining Prolog Databases, *Software Practice Experience* 21:603–624 (1991).
24. Lamma, E., Mello, P., and Natali, A., An Extended Warren Abstract Machine for the Execution of Structured Logic Programs, *J. Logic Programming* 14:187–222 (1992).
25. Mello, P. and Natali, A., Extending Prolog with Modularity, Concurrency and Meta-Rules, *New Generation Comput.* 10:335–359 (1992).
26. Miller, D., A Logical Analysis of Modules in Logic Programming, *J. Logic Programming* 6:79–108 (1989).
27. Montiero, L. and Porto, A., Contextual Logic Programming, in: *Proc. 6th Int. Conf. Logic Programming*, MIT Press, Cambridge, MA, 1989, pp. 284–302.
28. Miyachi, T., Kunifuji, S., Kitakami, H., Furukawa, K., Takeuchi, A., and Yokota, H., A Knowledge Assimilation Method for Logic Databases, in: *Proc. 1984 Int. Symp. Logic Programming*, IEEE Computer Society Press, Washington, DC, 1984, pp. 118–125.
29. Nakashima, K., Knowledge Representation in Prolog/KR, in: *Proc. 1984 Int. Symp. Logic Programming*, IEEE Computer Society Press, Washington, DC, 1984, pp. 126–130.
30. Nadathur, G., Jayaraman, B., and Kwon, K., Scoping Constructs in Logic Programming: Implementation Problems and Their Solution, *J. Logic Programming* 25:119–161 (1995).
31. *Quintus Prolog Reference Manual*, Quintus Computer Systems, 1985.
32. des Rivieres, J., Meta-Level Facilities in Logic-Based Computational Systems, in: *Proc. Workshop on Meta-Level Architectures and Reflection*, Alghero-Sardinia, Italy, 1986.
33. Roy, P. V., A Prolog Compiler for the PLM, Master's Thesis, University of California, Berkeley, 1984.

34. Russell, S., The Complete Guide to MRS, Knowledge Systems Laboratory Report KSL-85-12, Stanford, 1985.
35. Safra, M. and Shapiro, E., Meta-Interpreters for Real, in: E. Shapiro (ed.), *Concurrent Prolog*, Vol. 2, MIT Press, Cambridge, MA, 1987, pp. 166–179.
36. Sterling, L. and Shapiro, E., *The Art of Prolog*, MIT Press, Cambridge, MA, 1986.
37. Sterling, L. S., Meta-Interpreters: The Flavors of Logic Programming?, in: *Proc. Workshop Deductive Databases Logic Programming*, Washington, DC, 1986, pp. 163–175.
38. Sterling, L. S. and Beer, R. D., Incremental Flavor-Mixing of Meta-Interpreters for Expert System Construction, in: *Proc. 3rd Symp. Logic Programming*, IEEE Computer Society Press, Washington, DC, 1986, pp. 20–27.
39. Sterling, L. S. and Lakhotia, A., Composing Prolog Meta Interpreters, in: *Proc. 5th Int. Conf. Symp. Logic Programming*, MIT Press, Cambridge, MA, 1988, pp. 386–403.
40. Sterling, L. S., A Meta-Level Architecture for Expert System, in: R. Maes and D. Nardi (eds.), *Meta-Level Architectures and Reflection*, North Holland, Amsterdam, 1988.
41. Warren, D. H. D., An Abstract Prolog Instruction Set, SRI Technical Report 309, 1983.
42. Weyhrauch, R. W., Prolegomena to a Theory of Mechanized Formal Reasoning, *Artif. Intell.* 13:133–170 (1980).
43. Weyhrauch, R. W., An Example of FOL Using Metatheory, in: *Proc. 6th Conf. Automated Deduction*, Springer-Verlag, New York, 1982.
44. Yalcinalp, U. and Sterling, L., An Integrated Interpreter for Explaining Prolog's Successes and Failures, in: H. Ambramson and M. H. Rogers (eds.), *Meta-Programming in Logic Programming*, MIT Press, Cambridge, MA, 1989, pp. 191–204.