Meta-Interpreters and Expert Systems

Roman Barták[*] & Petr Štěpánek

Department of Computer Science

Faculty of Mathematics and Physics

Charles University

Malostranské nám. 25

Prague, Czech Republic

Abstract

Meta-programming is a well-known technique widely used in logic programming and artificial intelligence. Meta-interpreters are powerful tools especially for writing expert systems, in particular their inference machines. A classical approach to meta-interpretation is based on the syntactic definition of a meta-interpreter. A new approach, presented in this work, concentrates more on the meaning of the preposition *meta*. We used the structure of expert systems (problem solvers) for finding a general description of any meta-interpreter. We call this generalized meta-interpreter **a mega-interpreter**. The mega-interpreter is divided into two parts - the kernel and the extension. The kernel represents common features of interpreters while the extension is specific to a particular interpreter. Some comparison with a classical artificial intelligence search technique is also done.

[*] e-mail: bartak@ktisun1.ms.mff.cuni.cz

1 INTRODUCTION

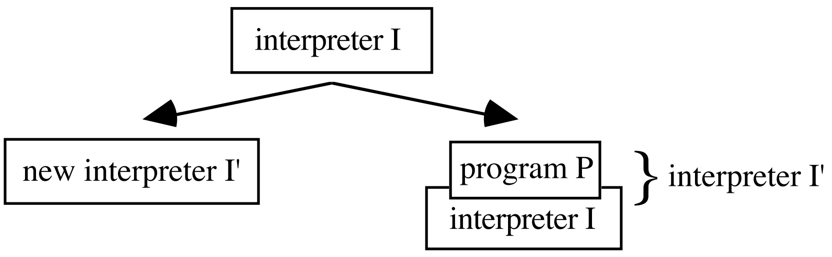
PROLOG is one of the languages used for writing rule-based expert systems and meta-interpreters are one of the widely used programming techniques for that purpose. Therefore, in what follows we use PROLOG as a basic language for our research. The reasons for popularity of meta-interpreters are the following: the meta-interpreters are relatively simple to use and understand, and, at the same time, they are very powerful. Many meta-interpreters have been written for special purposes ([8],[9]). In this paper, we concentrate on those meta-interpreters for the construction of expert systems. We try to find a uniform paradigm for writing such a type of meta-interpreters.

First, the classical definition of a meta-interpreter is reminded (the Section 2). Some examples of well-known meta-interpreters are also shown. We use these examples for finding another approach to meta-interpretation. This approach (described in the Section 3) is based on the meaning of the preposition *meta*. We introduce here also the notion of a mega-interpreter. The Section 4 describes two ways to the concept of the mega-interpreter. The key result of that section is the description of the mega-interpreter's structure. The *top-down* method has been used for that purpose.

The mega-interpreter is divided into two parts, the kernel and the extension. The mega-interpreter's kernel represents a general description of "all" interpreters. It has an imperative character. Specific information for a particular interpreter is encoded in the extension of the mega-interpreter. The extension should have a declarative character. Programming a new interpreter is then the same as writing the extension of a given kernel. One of the advantages of this approach is the shift from the imperative to the declarative character of interpreter's programming. Another advantage is described in the Section 6. It is a hierarchical structure of the mega-interpreter's kernel. Examples of the PROLOG source code for the kernel are also included. The *bottom-up* method was used for finding the PROLOG source code of the kernel. The last but two section of that paper is dedicated to comparison ideas of mega-interpreter with well-known technique used in artificial intelligence, search.

2 The Meta-Interpreters Classic

The classic definition of a meta-interpreter is based on the following idea. Let us have any programming language *L* and any interpreter *I* of that language. We want to write another interpreter *I'* of the same language *L*. There are at least two ways how to do it. First, we can write a completely new interpreter but it could be a very complicated and a time consuming process. Especially, if there is only a little difference between the interpreter *I* and the interpreter *I'*, this way is ineffective, even if you are a developer of the interpreter *I*. Therefore, the second approach is used more often. It is based on writing a program *P* in the language *L* that changes the behaviour of the interpreter *I* into the behaviour of the interpreter *I'*. The Picture 2.1 shows these two ways to writing a new interpreter.



Picture 2.1 (two approaches to writing a new interpreter)

The program *P* is called a meta-interpreter. If we want to write another interpreter, we simply write a new program *P'*. It is easier than programming a new interpreter. Here is the definition of the meta-interpreter.

DEFINITION [7]:

A **meta-interpreter** of a given programming language (or its subset) is an interpreter of that language (or the subset) that is written in the interpreted language.

As you can see, it is a very simple definition that determines which interpreter is the meta-interpreter. It is the relation between the language of the interpreter and the interpreted language that makes the definition of the meta-interpreter. Therefore we speak about the *syntactical basics* of this definition.

There is a philosophical problem with the definition. We have to know what is it an interpreter. Is it a program or a machine or both? It must be a program because the meta-interpreter, the special case of an interpreter, is "*written in the interpreted language.*" However, where the machine was lost? The program is just a text, a code. It cannot do anything, it cannot interpret anything. It is like a word. The word can not hurt, but saying the word can. To say a word we need a person, to interpret a program we need a machine. If we speak about an interpreter, we assume that there is a machine "under" the interpreter. However speaking about the meta-interpreter, we assume that there is a machine with an appropriate interpreter under the meta-interpreter (see the Picture 2.1). Therefore, there is a difference between the interpreter and the meta-interpreter. The classical definition does not consider this difference when it says "*a meta-interpreter is an interpreter.*"

Let us remind some examples of well-known meta-interpreters now. PROLOG is a suitable language for writing the meta-interpreters and generally the meta-programs. A meta-program is a program that works with another program as data. The meta-interpreter is a special case of the meta-program. The main advantage (for purposes of meta-programming) of PROLOG is the same structure of the program and the operated data. Because the code of the program represents data for the interpreter, it is easy to write the meta-interpreter. LISP also has a similar feature. The first example (the Program 2.1) shows the simplest meta-interpreter for PROLOG programs. It only calls a standard PROLOG interpreter.

```
solve(Goal):-call(Goal)
```

Program 2.1

It is a meta-interpreter according to the definition, but there is not any visible advantage of using it instead of standard PROLOG interpreter.

Now, we can write a similar meta-interpreter (the Program 2.2) with new features added to the standard interpreter of PROLOG.

```
        solve(Goal,Result):-
                prepare_goal(Goal,Goal_for_PROLOG),
                call(Goal_for_PROLOG),
                customize_solution(Goal,Goal_for_PROLOG,Result).
```

Program 2.2

This program makes a basic frame of the first level kernel of the mega-interpreter (the Section 4.2). Predicates `prepare_goal` and `customize_solution` are so called *user defined predicates*. The user can define them and so he can determine the resulting behaviour of the meta-interpreter. We shall call bellow the code of the user defined predicates – the extension of the mega-interpreter's kernel. The Program 2.3 shows an example of the definitions of the user defined predicates. The Program 2.2 (the kernel) with the Program 2.3 (the extension) solve PROLOG goals without binding free variables.

```
    prepare_goal(Goal,Goal_for_PROLOG):-
            copy_term(Goal,Goal_for_PROLOG).
    customize_solution(Goal,Goal_for_PROLOG,Result):-
            Result=[the,solution,of,the,goal,Goal,is,Goal_for_PROLOG].
```

Program 2.3

The power of this particular simple meta-interpreter is now evident.

The following meta-interpreter is a well-known PROLOG meta-interpreter. It is written in another level of abstraction (called clause reduction level). Meta-interpreters on that level make explicit the choice of clauses being used to reduce a goal, and the choice of literal to generate the resolvent. Unification and backtracking are handled implicitly, relying upon the behaviour of PROLOG [8]. Most other meta-interpreters are derived by making extension of this basic form. According to the obvious analogy with ice cream flavors [6] this meta-interpreter is called *vanilla*. Here it is.

```
solve(true).
solve((A,B)):-
          solve(A),
          solve(B).
solve(Goal):-
          clause(Goal,Body),
          solve(Body).
```

Program 2.4 (the vanilla meta-interpreter)

The following meta-interpreter is derived from the vanilla one. This derivation is very simple but resulting behaviour is dramatically changed. The meta-interpreter (the Program 2.5) solves any goal using strategy “from right to left” instead of standard PROLOG strategy “from left to right.” It is an another example of meta-interpreter‘s power and elegance.

```
solve(true).
solve((A,B)):-
          solve(B),
          solve(A).
solve(Goal):-
          clause(Goal,Body),
          solve(Body).
```

Program 2.5 (the vanilla with goal order changed)

The next program is another variant of the vanilla meta-interpreter. The mechanism of the interpreter was not changed but new information was added to the output of this program, namely the “proof” of successful answer.

```
solve(true,fact).
solve((A,B),(ProofA,ProofB)):-
                solve(A,ProofA),
                solve(B,ProofB).
solve(Goal,Goal-ProofB):-
                clause(Goal,B),
                solve(B,ProofB).
```

Program 2.6 (the vanilla with proofs)

The Program 2.6 makes a basic frame of other meta-interpreters, those are used for finding explanations in expert systems being programmed in PROLOG.

Now, there is a time for another definition. We spoke about different abstraction levels during presentation of the vanilla meta-interpreter. These levels have their own names.

DEFINITION [7]:

The **granularity** of the meta-interpreter is the level of access to the computation. The higher level means the finer granularity (the vanilla meta-interpreter has finer granularity than the simple meta-interpreter from the Program 2.1).

The level of abstraction (the granularity) is important for classifying the meta-interpreters. The meta-interpreters with fine granularity are able to change more the mechanism of the interpreter than those ones with rough granularity. Mostly the fine granularity means slower interpretation because the meta-interpreter must be interpreted by an interpreter – a core interpreter of a given programming language (see the Picture 2.1). Therefore the key problem in choosing the granularity is finding necessary granularity with respect to the speed of the interpretation.

Another problem with the meta-interpreters is doubling the space. It means that we use two interpreters (the core interpreter and the meta-interpreter) to simulate another interpreter and therefore many procedures are doubled (they exist in the core interpreter and in the meta-interpreter too). It also causes slow down of the computation. Compilation solves the problems with speed but the compiled meta-interpreter is not a meta-interpreter according to the definition.

3 NEW APPROACH TO META-INTERPRETATION

The following example shows why we prefer a different view on meta-interpretation. It is a program written in PROLOG, that simulates the computations of a finite automaton. We call this program (the Program 3.1) an interpreter because it interprets some code, namely the description of any finite automaton. It is not a meta-interpreter according to the definition mentioned above (there is a difference between „the interpreted language”, i.e., the description of the finite automaton, and the language of the interpreter, i.e., PROLOG).

```
solve(Q,[]):-
          final_state(Q).
solve(Q,[H|T]):-
          rule(Q,H,NewQ),
          solve(NewQ,T).
```

Program 3.1 (the finite automaton simulator)

As you can see, the Program 3.1 has a similar structure as the programs presented above, namely, the vanilla meta-interpreter (the Program 2.4). The possibility of making changes to the mechanism of the interpreter and making derivative programs is also saved.

The next program, the Program 3.2, has been developed using the same derivation step as in the derivation the Program 2.6 (the vanilla with proofs) from the Program 2.4 (the vanilla meta-interpreter).

```
solve(Q,[],Q):-
          final_state(Q).
solve(Q,[H|T],Q-Rest):-
          rule(Q,H,NewQ),
          solve(NewQ,T,Rest).
```

Program 3.2 (the finite automaton with proofs)

The output of program’s computation gives a solving succession of states (it is an equivalent to the proof).

The above examples demonstrate that the main advantage of the meta-interpreters consists in an easy and simple access to the mechanism of the interpreter. One might be tempted to say: “A meta-interpreter is an interpreter that makes possible **an easy access to its own mechanism**.” However, it is not a definition of the meta-interpreter (what is it easy?) but we think it describes more accurately the character of meta-interpreters (a feature that users like on them). The core of that “definition” is the relation between the interpreter and the mechanism of this interpreter.

As we have already mentioned above, our description of the concept of the mega-interpreter will be based on the meaning of the preposition *meta*. It is a frequently used preposition (a meta-program, a meta-theory, a meta-variable …) that means “something over the object level” [1]. The meta-theory is a theory over another theory; **the meta-interpreter is an interpreter of another interpreter**. It means that the meta-interpreter is an interpreter that interprets a code (i.e., a description) of another interpreter (will be explained in the Section 4.1).

In the following section we try to combine ideas of these two pseudo-definitions into one compact form. The result will be called **a mega-interpreter**. Why mega? Because the word *mega*[1] means *huge* and our approach can be used for a huge amount of different interpreters.

4 THE STRUCTURE OF THE MEGA-INTERPRETER

Now, let us compare our two pseudo-definitions.

A meta-interpreter is an interpreter (a common part of both pseudo-definitions)

- that enables an “easy access” to the mechanism of the interpreter, /a/
- that interprets a code (a description) of some interpreter. /b/

The pseudo-definition /a/ is a consequence of the definition of the meta-interpreter. A meta-interpreter written in the interpreted language enables an “easy access” to the mechanism of the

[1]from the Greek words megas and megalé

interpreter in most cases. The part /a/ is user oriented because users like everything what is easy. However, what does it mean “easy”?

The pseudo-definition /b/ is based on the meaning of the preposition *meta*. It could be a definition not only a pseudo-definition, if we define what is it an interpreter and what is it a description of an interpreter. However, there is another problem with the pseudo-definition /b/. The pseudo-definition /b/ of the meta-interpreter depends on the interpreted program not on the interpreter itself. Therefore, every interpreter is a meta-interpreter if it interprets some description of another interpreter and, at the same time, it is only an interpreter if it interprets a “normal” program, not the description of the interpreter.

We shall show that there is **a big gap** between the pseudo-definition /b/ and the classical definition of the meta-interpreter. Let us have any interpreter *I* of a given programming language *L*. The meta-interpreter of the same programming language *L* is a program *P* written in the language *L* that interprets the same codes (programs) as the interpreter does (i.e., programs written in *L*). If we want to modify the mechanism of the meta-interpreter *P*, we must change the whole meta-interpreter (compare the Program 2.4 – vanilla – with the Program 2.5 – vanilla with changed goal order).

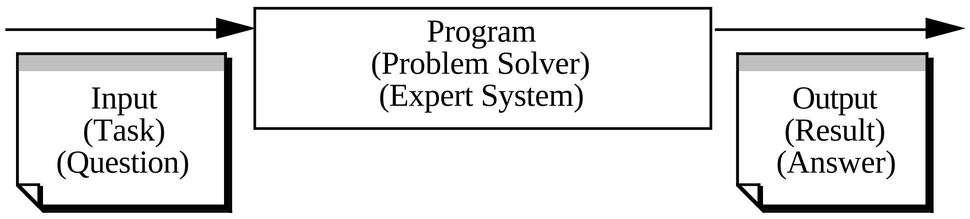
On the other hand, it follows from the pseudo-definition /b/ that we can call the interpreter *I* a meta-interpreter (let us call it rather the b-meta-interpreter) if it interprets the program *P* that we can call the description of the interpreter. We need not change the b-meta-interpreter (in this particular case the interpreter *I*), if we want to modify the mechanism of the b-meta-interpreter. We have to change only the data, the code (i.e., the description) of the interpreter. In this particular case, the description of the interpreter is the program *P*. However, it is not a typical example of using the pseudo-definition /b/. It shows that the pseudo-definition /b/ includes the classical approach too. Changing the data is easier than changing the program. Therefore, making changes to the mechanism of the b-meta-interpreter should be easier. The following table shows notions from the classical approach and the pseudo-definition /b/ respectively.

	classical approach	**pseudo-definition /b/**
program P	meta-interpreter	description of interpreter
interpreter I	interpreter	meta-interpreter

We explain the approach, based on the pseudo-definition /b/, more accurate by using simple pictures in the following section. Of course, we try to overcome the gap by introducing a new notion, a mega-interpreter.

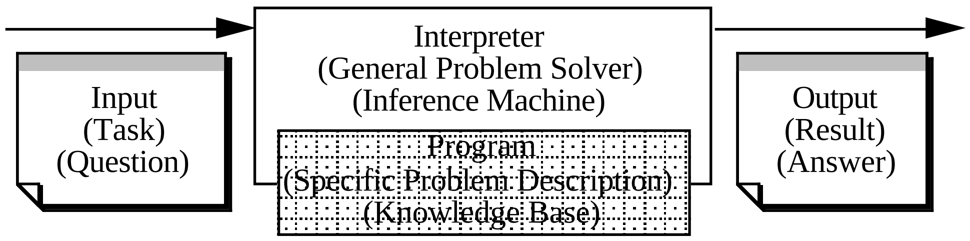
4.1 PROBLEM SOLVING AND META-INTERPRETERS (TOP-DOWN METHOD)

We will use a different method here to achieve the same result as above, namely the meta-interpreter as a program that interprets a code of some interpreter. We will call the resulting meta-interpreter **a mega-interpreter**. The following sequence of pictures describes a top-down way to construct the structure of the mega-interpreter.



Picture 3.1 (program)

The Picture 3.1 shows the structure of any program (any problem solver, any expert system) as a black box.



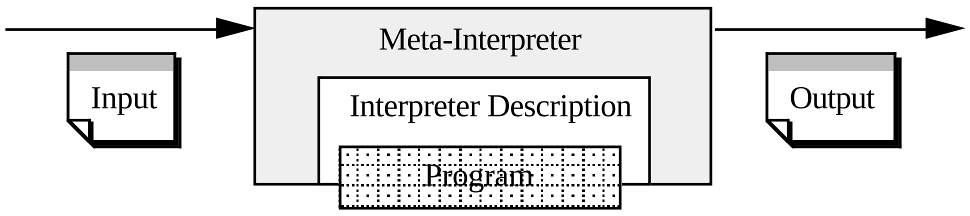
Picture 3.2 (structure of the program)

The Picture 3.2 also shows the structure of any program (any problem solver, any expert system) but in more detail. The program has two parts: an interpreter and a program description.

Let us call the step from the Picture 3.1 to the Picture 3.2 **the specification step applied to a given program**. The specification step describes more formally our previous discussion of using the preposition *meta*.

The Picture 3.2 corresponds to the current situation in expert systems research. An empty expert system contains an inference machine and a knowledge base. A particular expert system is defined by a particular knowledge base. It is not possible to change the inference machine but it is possible to change the knowledge base. Sometimes, we also need to change the inference mechanism of the expert system. Then we can use the following approach.

Let us perform the specification step to the interpreter now.

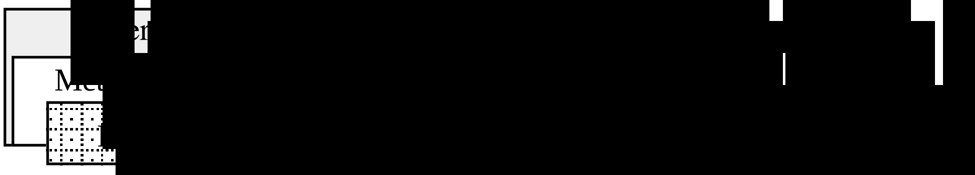


Picture 3.3 (new approach to meta-interpreters)

The Picture 3.3 shows the result. An interpreter is divided into two parts: a meta-interpreter and a description of the interpreter. In terminology of expert systems, we can speak about a general inference machine (a meta-inference machine) and about a description of a particular inference machine. To be user friendly, we need easy descriptions of the interpreter and the inference machine respectively. It will be also good if the program and the knowledge base could influence the interpreter mechanism and the inference mechanism respectively through their descriptions. The Picture 3.3 also corresponds to the pseudo-definition /b/.

Of course, we can now continue in performing specification steps and the result will be the meta-meta-interpreter with the description of the meta-interpreter, etc. There is no visible gain of doing it.

Now, we shall concentrate on the problem how to overcome the terminology variance. We want to be consistent with the classical approach to meta-interpretation but, at the same time, we would like to generalize the concept of the meta-interpreter. There is a difference between the classical approach and our approach. The classical one is based on specialization while we use generalization. We think that [redacted] classical one (see the discussion of *the bi*[redacted] between the concepts of these two ap[redacted] Picture 3.4).



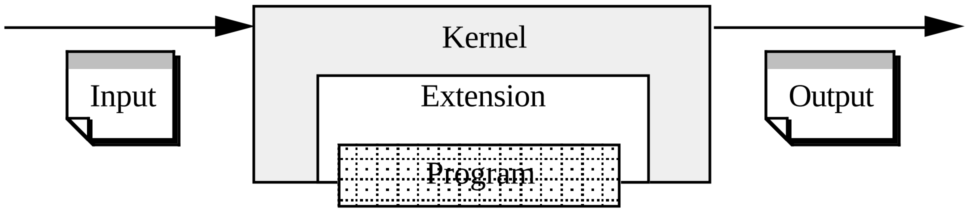
Picture 3.4 (comparison of [redacted]oaches to meta-[redacted]rpretatio[redacted]

There is a simple solution to this terminology problem. It is based on renaming the concepts of the new approach:

the meta-interpreter	$\to$	the kernel of the mega-interpreter
the description of the interpreter	$\to$	the extension of the kernel.

We can now describe a mega-interpreter by simple equation:

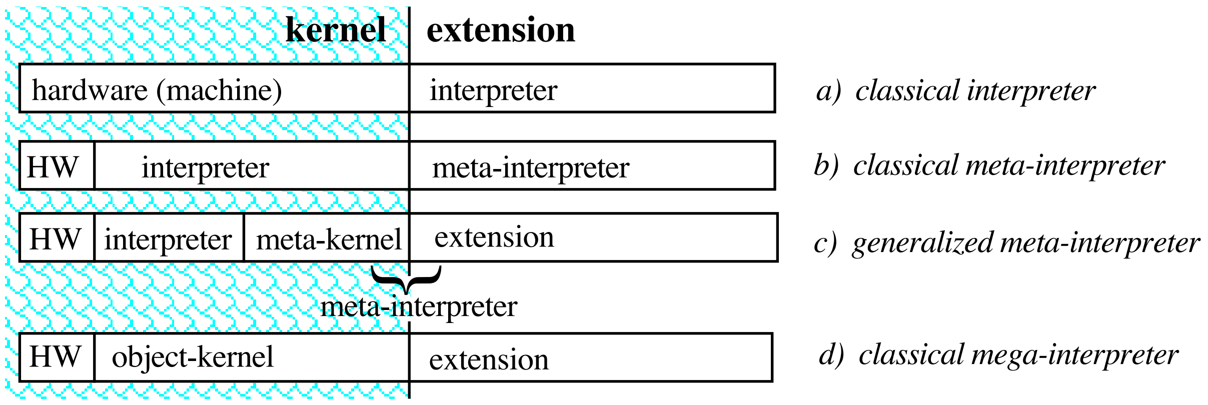
mega-interpreter = kernel + extension.



Picture 3.5 (structure of the mega-interpreter)

The Picture 3.5 displays the structure of the mega-interpreter. It is possible to write a fixed kernel and complete it by any extension to various interpreters. We shall explain roles of all the parts shown in the Picture 3.5. The **program** describes how to solve a particular problem. It is written in a given programming language *L*. The **extension** represents description of a particular interpreter (a PROLOG interpreter, a LISP interpreter, etc.) of the given programming language *L*. The **kernel** is a general machine (program) for interpreting programs. The kernel and the extension make a particular interpreter of the given programming language.

Now, we shall show what are the advantages of this approach in comparison to the classical one. First, the structure of the mega-interpreter is scalable. The Picture 3.6 explains what does it mean.



Picture 3.6 (scalable structure of the mega-interpreter)

Note that the parts a) and b) represent current approach to interpretation. We spoke above about some problems of these two approaches, in particular about complicated changes (the part a) and about inefficiency and doubling (the part b). The common problem consists in the fact that the extension has an imperative character and therefore it makes it more complicated to change the mega-interpreter. The part b) also shows that the classical approach to meta-interpretation can be expressed in terms of the mega-interpreter.

The parts c) and d) are more interesting. We shall speak about the part c) later in this paper (the Section 4.2). It is similar to a classical meta-interpreter and therefore it inherits some bad features of meta-interpreters, in particular inefficiency and doubling the space. The part d) represents a classical structure of the mega-interpreter with the kernel written in the object (machine-oriented) language and the extension written in any high-level language. We will prefer to write the extension in the interpreted language because then it will be easy to change the interpreter’s mechanism and the interpreted program can influence the interpreter’s mechanism through the extension. The classical mega-interpreter (the part d) of the Picture 3.6) removes the problems with inefficiency and doubling, because there is only one interpreter, hidden in the object kernel, that can be influenced through the extension.

We shall concentrate on writing the mega-interpreters with the structure similar to the part c) of the Picture 3.6 because it is easy to write the meta-kernel and the extension in a high-level language. By compiling the meta-kernel we can simple get an object-kernel and so remove the inefficiency and doubling. We speak about the object-kernel and the meta-kernel because they are only parts of the kernel. The object-kernel is written in an object language while the meta-kernel is written in the interpreted language.

Now we can give a definition of a general mega-interpreter.

DEFINITION:

A **mega-interpreter** of a given programming language is an interpreter of that language that is set up of two parts: the kernel and the extension. The extension is written in the superset of the interpreted language.

The power of a mega-interpreter is in the separation of the kernel and the extension. Common features of interpreters can be hidden in the kernel, while the extension contains specific information for a particular interpreter. This information is encoded in the interpreted language or in its superset. Therefore, there is an easy access to the mechanism of the interpreter.

The definition of the mega-interpreter covers all parts of the Picture 3.6. For example, a LISP machine (the kernel) and a LISP interpreter (the extension) written in LISP make a mega-interpreter. Its structure corresponds to the part a) of the Picture 3.6. All PROLOG meta-interpreters make the extensions of the PROLOG mega-interpreters. These two examples are not typical for the mega-interpreters because of the imperative character of the extension. We prefer a declarative character of the extension because it is easier to write a declarative description of the interpreter than an imperative one. Therefore we also prefer the meta-interpreters consisting of the meta-kernel with an imperative character and the extension with a declarative character (the part c) of the Picture 3.6). We call the mega-interpreter with the declarative extension an **easy mega-interpreter**.

In the following text we shall speak only about the easy mega-interpreters those have the same structure as in the part c) of the Picture 3.6. The meta-kernel will be called here the kernel because, clearly, there are also any machine and any PROLOG interpreter in the mega-interpreter's kernel.

In the following section, we try to find the PROLOG code of the meta-kernel and of the interface to the extension respectively. We will concentrate on the hierarchical structure of the program, as well.

4.2 From Meta-Interpreters To The Code Of The Kernel (Bottom-Up Method)

This section contains some other examples of PROLOG meta-interpreters. We use them to identify common features that could be encoded and thus grasped by the kernel.

Let us start with the Programs 2.1 - 2.6. There can be identified a common feature that we call the **Zero Level Kernel**. In fact, it means an empty kernel and therefore the case b) from the Picture 3.6. The Zero Level Kernel is specified by the following predicate:

`solve(Goal)` or more generally by the predicate `solve(Goal,Result)`.

Many programmers use the predicate `solve` while programming meta-interpreters or interpreters. The arguments of such predicates can be mostly separated into two groups: input (goal) and output (result) arguments. The Zero Level Kernel accords with a current situation in programming of meta-interpreters.

The following two meta-interpreters are also well-known. We use them to demonstrate some common features that will be encoded in the **First Level Kernel** of the mega-interpreter. The programs comply with the Zero Level Kernel (input and output arguments, output exists for every input).

```
{1a} solve(true,0).
{1b} solve((A,B),N):-
         solve(A,NA),
         NA>=0,
         solve(B,NB),
         NB>=0, N is NA+NB.
{1c} solve(Goal,N):-
         clause(Goal,Body),
         solve(Body,NB),
         NB>=0, N is NB+1.
{1d} solve(_,-1).
```

Program 4.1

```
{2a} solve([],0).
{2b} solve([A|Rest],N):-
        clause(A,B),
        append_to_goal(B,Rest,NewGoal),
        solve(NewGoal,N1),
        N1>=0,
        N is N1+1.
{2c} solve(_,-1).
```

Program 4.2

Both the Programs 4.1 and 4.2 solve "similar" PROLOG goals but there is a difference in the structure of the goal between them. The Program 4.2 replaces the conjunction of primitive goals, that is used by the Program 4.1, by the list of these primitive goals. However this difference is not significant for our research. Both the programs return the number of clauses being used for solving the goal as their output and `-1` if there is not any (other) solution. This number corresponds to the complexity of the computation.

Let us concentrate on the structures of both the programs. The *first* clause of both the programs (the clauses `1a` and `2a`) is used for solving an *empty goal*. In the Program 4.1, the empty goal corresponds to the term `true`, in the Program 4.2, it corresponds to the empty list (therefore "the empty goal"). The empty goal has the form that could be described by the sentence: "There is nothing to solve, everything has already been done." The *last* clause of both the programs (the clauses `1d` and `2c`) is the so called "catch all" clause. The "catch all" clause is used for returning a solution in the form: "Program is not able to find a (another) solution."

Let us call the conjunction or the non-empty list of primitive goals a *complex goal*. The complex goal is an opposite concept to the empty goal. Sometimes, the complex goal consists of only a one primitive goal (in the Program 4.1). The most important difference between the Programs 4.1 and 4.2 is in the method of solving the complex goals. The Program 4.1 uses two clauses for that purpose, namely the clauses `1b` and `1c`. The clause `1b` divides the conjunction into two parts and then solves each part independently. The clause `1c` transfers the primitive goal `Goal` into another goal `Body` using a certain clause and then solves the new goal `Body`.

The Program 4.2 uses only a one clause, namely the clause `2b`, for solving a complex goal. The process of solving a complex goal can be described in five steps there. Therefore we can speak about the *Five Step Principle* of solving a complex goal. First, the primitive goal `A` is selected from the complex goal `[A|Rest]`. This step also includes the hidden test whether the goal is really complex (the goal has the form `[A|Rest]` and therefore it must be complex). The first step is implicit and it is hidden in the head of the clause `2b`. In the second step, the selected primitive goal `A` is transferred to another goal `Body` using a certain clause. Then, the new goal `Body` is combined with the rest of the original complex goal (`Rest`) and a new goal (`NewGoal`) is made (the third step). The new goal is solved using the same mechanism, namely the predicate `solve`. Finally, the solution `N1` of the new goal is transferred to the solution `N` of the original goal.

Let us summarize the **Five Step Principle** now:

1) select the primitive goal A from the complex goal G (including test whether the goal G is a complex one)
2) transfer the selected primitive goal A into another goal B
3) combine the transferred goal B with the rest of the original complex goal G and make a new goal NG
4) solve the new goal NG
5) transfer the solution of the new goal NG into the solution of the original goal G.

The Five Step Principle can be also found in the Program 3.1 (the finite automaton simulator) and it is therefore more general. This principle makes a basic idea of the First Level Kernel.

We can also find the Five Step Principle in the Program 4.1 but it is spread over the clauses `1b` and `1c`. Now, we shall show that the Program 4.1 can be re-written according to the Five Step Principle. The resulting program (the Program 4.3) makes the core of the First Level Kernel. Therefore, we shall use the word “task” instead of “goal” because it is more general.

```
solve(true,0).
solve(Task,N):-
    select_subgoal(Task,Goal,Rest),
    clause(Goal,Body),
    make_task(Body,Rest,NewTask),
    solve(Body,N1),
    N1>=0, N is N1+1.
solve(_,-1).
```

```
select_subgoal((A,B),Goal,[B|T]):-
     select_subgoal(A,Goal,T).
select_subgoal(Goal,Goal,[]):-
     Goal\=(_,_).

make_task(Goal,[B|T],NewTask):-
     make_task(Goal,T,NewA),
     and(NewA,B,NewTask).
make_task(Goal,[],Goal).
```

Program 4.3

The Program 4.3 is not fully equivalent to the Program 4.1, because it does not process goals like `(true,(true,true))` while the Program 4.1 does. This difference can be simply removed by a more complicated definition of the predicate `select_subgoal` but it is not important for purposes of this work.

Now, it will be easy to write the program for the First Level Kernel. It is based on the generalized definition of the predicate `solve` from the Program 4.3.

```
solve(Task,Result):-
     empty_goal(Task,Result).
solve(Task,Result):-
     select_subgoal(Task,Goal,Frontier),
     expand_goal(Goal,ExpandedGoal,Rule),
     make_task(Frontier,ExpandedGoal,NewTask),
     solve(NewTask,SubResult),
     customize_solution(Frontier,Rule,SubResult,Result).
solve(Task,Result):-
     rest_solution(Task,Result).
```

Program 4.4 (the First Level Kernel)

The first clause of the Program 4.4 is used for solving empty goals, the last clause of the same program represents a “catch all” clause. The Five Step Principle is encoded in the second clause of the Program 4.4. Each goal in the body of this clause corresponds to the step from the Five Step Principle.

Some new notions appear in the Program 4.4, namely the Frontier and the Rule. The variable `Frontier` contains information about the selection of the subgoal (for example, the rest of the task). This information is used during the process of making a new task (see the Program 4.3 where the variable `Rest` substitutes the Frontier). Sometimes, we will need to add some complementary information to the Frontier during making a new task and this information will be used during customizing of the solution (see the Program 4.7 in the following section). The variable `Rule` contains information about goal transformation, for example the description of the rule (the clause) used for the transformation. This information is used during customizing of the solution too.

Now, it is time to say a few words about the interface between the kernel and the extension. Because we have selected PROLOG as a basic language for writing extensions, the interface consists of the list of predicates. These user defined predicates make hooks in the kernel where the user can hang the procedures that modify the behaviour of the kernel. The definition of (the programs for) these predicates forms the extension.

The interface to the Zero Level Kernel is simple. It consists of the only predicate `solve`. Therefore almost every PROLOG meta-interpreter makes an extension of the Zero Level Kernel. The interface to the First Level Kernel is more complicated. It consists of the following predicates: `empty_goal`, `select_subgoal`, `expand_goal`, `make_task`, `customize_solution` and `rest_solution`. Some examples of extensions of the First Level Kernel will be given in the following section.

4.3 Some Examples Of Extensions And The Second Level Kernel

We start this section with a quite complicated example of extension of the First Level Kernel. This extension (the Program 4.5) and the First Level Kernel (the Program 4.4) fully describe the mega-interpreter that simulates behaviour of the Program 2.6 (the program that computes proofs)[2]. We also use this extension for presenting a possible structure of the

[2]goals in forms like `(true,(true,true))` are not processed by this extension

Frontier. It is a good example of extension where we need to add some information to the Frontier during making a new task and use this information during the process of customizing solution. The First Level Kernel (the Program 4.4) and this extension (the Program 4.5) shall make the core of the Second Level Kernel.

```
empty_goal(true,fact).

select_subgoal((A,B),G,[(B,_)|T]):-
       select_subgoal(A,G,T).
select_subgoal(G,G,[]):-
       G\=(_,_),G\=true.

expand_goal(A,B,clause(A,B)):-
       clause(A,B).

make_task([(B,Ch)|T],Goal,Task):-
       make_task(T,Goal,A),
       and(A,B,Task),
       if_then_else(A=true,Ch=yes,Ch=no).
make_task([],Goal,Goal).

customize_solution([(_,no)|T],Rule,(SProofA,ProofB),(ProofA,ProofB)):-
       customize_solution(T,Rule,SProofA,ProofA).
customize_solution([(_,yes)|T],Rule,ProofB,(ProofA,ProofB)):-
       ProofB\=failed,
       customize_solution(T,Rule,fact,ProofA).
customize_solution([],clause(A,B),ProofB,A-ProofB):-
       ProofB\=failed.

rest_solution(_,failed).
```

Program 4.5

The resulting mega-interpreter consisting of the Programs 4.4 and 4.5 is more complicated than the original description of the interpreter in the Program 2.6. It is clear because this new program (consisting of the Programs 4.4 and 4.5) has finer granularity than the Program 2.6. The subgoal selection, making a new task and customizing solution is handled implicitly in the

Program 2.6. It means that we have no information about other subgoals during solving the subgoal there.

We can also use the same kernel (the Program 4.4) without any change for writing a completely different mega-interpreter. We have to do only one thing – write a new extension. The following program is an example of using the same kernel for a quite different interpreter. It is the simulator of a finite automaton similar to the Program 3.1.

```
empty_goal([]-Q,accept):-
      final_state(Q).

select_subgoal([H|T]-Q,H-Q,T).

expand_goal(A-Q,NewQ,not_used):-
      rule(Q,A,NewQ).

make_task(T,Q,T-Q).

customize_solution(_,_,accept,accept).
rest_solution(_,no).
```

Program 4.6

The interpreted language consists of only two predicates (if we do not include the predicates of the interface that are not used in the bodies of the clauses): `final_state` and `rule`. It corresponds to the classical description of a finite automaton (the finite automaton is fully described by the set of final states and the set of transformation rules).

However, there is a difference between the Programs 4.5 and 4.6. The Program 4.6 is rather declarative while the Program 4.5 is imperative. Therefore the First Level Kernel is satisfactory for the simulator of a finite automaton, but it is not suitable for the PROLOG mega-interpreter (the extension should have a declarative character). We introduce the Second Level Kernel to save the declarative character of extension.

There are three predicates of imperative character in the Program 4.5, namely, `select_subgoal`, `make_task` and `customize_solution`. The structure of these predicates is determined by the structure of the Frontier and vice-versa. We have chosen the structure of the Frontier as simple as possible. It is a list of pairs. This list (the Frontier) arises during a process of subgoal selection. The first parts of the pairs of this list are instantiated there while the second parts stay free (see the Programs 4.5 and 4.7). The second parts of the pairs can be instantiated during the process of making a new task where the first parts are already used (see again the Program 4.5). Finally, the Frontier can be used during customizing the solution (like in the Program 4.5). The length of the Frontier is determined by the "depth" of the task. The depth of the task is equal to the number of steps used to find a subgoal of this task. Therefore, the depth of the task is determined by the structure of the task and by the strategy of subgoal selection.

We introduce here two new concepts, namely the meta task and the simple task. We can directly select the subgoal from the *simple task* using the predicate `custom_goal_selection`. Therefore, the structure of the simple task is invisible to the extension, i.e., to the meta-level, and the depth of the simple task is equal to one. The opposite concept to simple task is the *meta task*. The meta task can be fallen into some "independent" subtasks using the predicate `goal_selection`. Therefore, the structure of the meta task is visible to the extension, i.e., to the meta-level, and the depth of the meta task is at least two. For example, the conjunction of goals is a meta task while the primitive goal is a simple task in PROLOG mega-interpreter.

Now, we can write the PROLOG code of the Second Level Kernel.

```
select_goal(Task,Goal,[(S,_)|T]):-
      meta_task(Task),
      goal_selection(Task,SubTask,S),
      select_goal(SubTask,Goal,T).
select_goal(Task,Goal,[(S,_)]):-
      simple_task(Task), /* not meta_task(Task) */
      custom_goal_selection(Task,Goal,S).

make_task([(S,Ch)|T],Goal,Task):-
      make_task(T,Goal,SubTask),
      combine_task(S,SubTask,Task,Ch).
make_task([],Goal,Goal).

customize_solution([F|T],Rule,SubSol,Sol):-
      decombine_solution(F,SubSol,Sol1,Sol2),
      customize_solution(T,Rule,Sol1,SSol1),
      combine_solution(F,SSol1,Sol2,Sol).
customize_solution([],Rule,SubSol,Sol):-
      combine_rule_solution(Rule,SubSol,Sol).
```

Program 4.7 (part of the Second Level Kernel)

The Second Level Kernel consists of the First Level Kernel (the Program 4.4) and of the Program 4.7. Because of the fixed structure of the Frontier, the Second Level Kernel is suitable for interpreting languages that satisfy the following criterion:

“The process of subgoal selection must fully determine the processes of making a new task and customizing solution.”

PROLOG is an example of language that satisfies this criterion (mostly, we do not need implicitly customize the solution). Of course, the languages that do not satisfy this criterion can be also interpreted by the Second Level Kernel, but they are not supported. It means that the mega-interpreters of these languages will not take advantage of all the user defined predicates of the Second Level Kernel.

The Second Level Kernel can be used for a wide range of mega-interpreters. It is easy to prove that every mega-interpreter, that can be written using the Second Level Kernel, can also be written using the First Level Kernel and vice-versa. We also hope that the Second Level Kernel is suitable for writing the inference machines of the rule-based expert systems. The Second Level Kernel then represents a shell of the inference machine.

In the following section we shall show that search as one of the general techniques used in artificial intelligence and in construction of expert systems can be easily implemented using the ideas of mega-interpreters.

5 MEGA-INTERPRETERS VERSUS SEARCH

Search is the most widely used technique in artificial intelligence. Both the programs playing chess or the expert systems use search as a basic algorithm for solving problems. We shall simulate search using the mega-interpreter now.

Let us have a directed acyclic graph. The main problem of search is to find a path from one node of this graph to another node. We shall use the simplified First Level Kernel to implement simple search on the platform of mega-interpreters. As we have mentioned above, the First Level Kernel is based on the Five Step Principle. Now, we reduce the first three steps of the Five Step Principle, namely subgoal selection, goal expansion and making a new task, into one step. By this way, the *Three Step Principle* appears.

The Three Step Principle:

1) transfer the task T into another task NT
2) solve the new task NT
3) transfer the solution of the new task NT into the solution of the original task T.

The Three Step Principle is suitable for solving simple structured tasks that can be directly transferred to other tasks. Therefore, we do not need to select a subgoal from the task. At the same time, we prefer the transfered task NT to be simpler than the original task T. However, this assumption is not forced there (in general, it is impossible to determine whether the task NT is simpler than the task T).

It is easy to implement the Three Step Principle in PROLOG. We can call the resulting program (the Program 5.1) the Half Level Kernel.

```
solve(Task,Result):-
     empty_goal(Task,Result).
solve(Task,Result):-
     transfer_task(Task,NewTask,Frontier),
     solve(NewTask,SubResult),
     customize_solution(Frontier,SubResult,Result).
solve(Task,Result):-
     rest_solution(Task,Result).
```

Program 5.1 (the Half Level Kernel)

The Half Level Kernel is appropriate for implementing some simple search algorithms. The following program (the Program 5.2) and the Half Level Kernel (the Program 5.1) represent the simplest search technique (called the depth-first search) on the platform of mega-interpreters.

```
empty_goal(Node,yes):-
         final_node(Node).
transfer_task(Node,NewNode,not_used):-
         not final_node(Node),
         edge(Node,NewNode).
customize_solution(not_used,yes,yes).
rest_solution(_,no).
```

Program 5.2 (the extension for search)

The Task corresponds to the initial node of the search here. The description of the graph (the edges) and the set of final nodes are represented by the “interpreted program.” Sometimes, we need to include the set of final nodes into the Task. In this particular case, it is possible to use the following extension (the Program 5.3). When the task becomes more complicated, it will be better to use the First Level Kernel or higher.

```
empty_goal(SNode-FNodes,yes):-
         member(SNode,FNodes).
transfer_task(SNode-FNodes,NewNode-FNodes,not_used):-
         not member(SNode,FNodes),
         edge(SNode,NewNode).
customize_solution(not_used,yes,yes).
rest_solution(_,no).
```

Program 5.3 (the second extension for search)

There are many other extensions that implement search on the platform of mega-interpreters. Some of them can implement search in the directed graphs with cycles, other ones can compute the path from the initial node to one of the final nodes. These simple examples show that it is easy to implement search using the mega-interpreters and, therefore, that it is easy to implement many AI programs, including expert systems, on the platform of mega-interpreters. However, many other approaches also enable easy implementation of search, but the advantage of using mega-interpreters is somewhere else.

What are the main benefits of using mega-interpreters for implementing search algorithms? The advantage of mega-interpreters is in separating the general part (the kernel) of the search algorithm from the specific one (the extension). This enables a construction of “empty” inference machines for expert systems (an inference machine represents a special case of search). An *empty inference machine* differs from an empty expert system[3]. For example, the mega-interpreter enables one to implement different techniques of handling uncertainty or different search strategies (i.e., different inference machines) using the same empty inference machine. Therefore, an empty inference machine is more general than an empty expert system and it can be used for wider range of different expert systems.

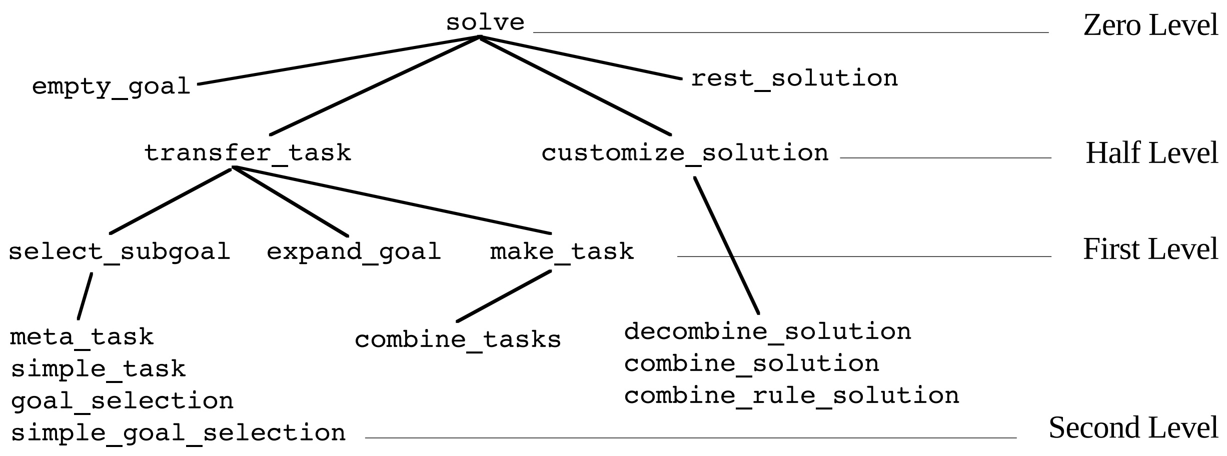
Another advantage of using mega-interpreter is the possibility to influence the inference mechanism through the extension. Therefore, the particular knowledge base can easily influence the inference mechanism depending on the problem. It is different from the drawback of “programming” knowledge base, because the extension has a declarative character and, therefore, the declarative character of the knowledge base is preserved.

[3] an empty expert system is an expert system without particular knowledge base i.e. containing only an inference machine

A more general benefit of mega-interpreters is in re-using of the kernel (see the Programs 5.2 and 5.3). This feature is similar to inheritance in OOP and it speeds up the development of new programs.

6 Final Notes On Mega-Interpreters

We spoke about three (four) different levels of the meta-kernel of the mega-interpreter for PROLOG. This hierarchical structure makes programming of (mega)interpreters more comfortable because the programmer may select an appropriate level of the meta-kernel. We can display the above mentioned hierarchy of PROLOG meta-kernels using the following picture.



Picture 6.1 (the hierarchy of kernels)

Each hierarchy level points out to the corresponding granularity. In most cases, the higher level means the finer granularity (the granularity also depends on the extension). However, when we use the mega-interpreters, the higher level does not imply the slow down of the computation and the doubling (see the end of the Section 2). We can compile the meta-kernel of the mega-interpreter into the object-kernel and by this way we can save the computation speed. It is recommended to use extensions as simple as possible and to hide “unnecessary ballast” in the kernel.

Now, let us summarize how to find the code, i.e., the level, of the meta-kernel. Let us have a given kernel and some extensions of this kernel that are too complicated, for example, the extensions have an imperative character. We try to find some common features of these extensions and, finally, we encode these common features into the higher level meta-kernel. We used the same way during construction of the First and the Second Level Kernels.

7 CONCLUSIONS

In this paper we have described a new approach to meta-interpretation. This approach is based on the definition of a mega-interpreter. The mega-interpreter preserves the positive features of meta-interpreters (i.e., an easy access to interpreter's mechanism) and, at the same time, it eliminates the slow down of the computation and doubling.

The idea of a mega-interpreter is based on the separation of a general part of an interpreter from a specific one. The mega-interpreter consists of two parts: the kernel and the extension. Thanks to the hierarchical structure of the kernel, the user can select the level (granularity) that best suits his or her needs without loss of speed typical for meta-interpretation. The hierarchical structure of the kernel can be also used for the classification of (mega)interpreters.

A uniform frame for writing (mega)interpreters helps programmers to concentrate on features of a particular (mega)interpreter without troubles with general principles of interpretation. The idea of mega-interpreter can also work as a consolidating element in the reflective programming. Our approach can help in composing interpreters or developing program modulants, enhancements and mutants [8] too.

Mostly, we have concentrated on using mega-interpreters for the construction of inference machines of expert systems and problem solvers. A comparison with search, a classical technique used for construction of expert systems, has therefore been done. The concept of mega-interpreter also arose originally from the research of using meta-interpreters for creation of expert systems. A notion of empty inference machine was introduced there. An empty inference machine can be extended to a specific inference machine by adding a particular extension. This process simplifies and speeds up the construction of a particular inference machine and also of the particular expert system.

We used PROLOG as a basic programming language for the research because it is an ideal language for writing meta-programs. However, the results can be applied to other languages too. We have also presented few example programs to show a practical usability of the idea of the mega-interpreters.

REFERENCES

[1] Abramson, H. and Rogers, M.H. (eds.), *Meta-Programming in Logic Programming*, The MIT Press, Cambridge, Massachusetts, 1989

[2] Barták, R., Meta-interpretation of Logic Programs (in Czech), Diploma Thesis, Charles University, Prague, 1993

[3] Clocksin, W.F. and Mellish, C.S., *Programming in Prolog*, Springer-Verlag, Berlin, 1981

[4] Nilsson, N.J., *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971

[5] Parsaye, K. and Chignell, M., *Expert Systems for Experts*, John Wiley & Sons, New York, 1988

[6] Sterling, L., Meta-Interpreters: The Flavors of Logic Programming?, in: *Proceedings of Workshop on foundation of Logic Programming and Deductive Databases*, Washington, 1986

[7] Sterling, L., Constructing Meta-Interpreters for Logic Programs, in: *Advanced School on Foundations of Logic Programming*, Alghero, Sardinia, Italy, september 1988

[8] Sterling, L. and Lakhotia, A., Composing Prolog Meta-Interpreters, in: *Proceedings of 5th International Logic Programming Conference*, Seattle, 1988

[9] Sterling, L. and Shapiro, E., *The Art of Prolog*, The MIT Press, Cambridge, Massachusetts, 1986