Extendible Meta-Interpreters

Roman Barták & Petr Štěpánek

Department of Theoretical Computer Science
Charles University
Malostranské náměstí 25
Praha 1, Czech Republic

e-mail: {bartak,stepanek}@ksi.ms.mff.cuni.cz
phone: +420-2 2191 4242
fax: +420-2 53 27 42

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 general and for writing their inference machines in particular. While the traditional approach to meta-interpretation is based on the syntactic definition of a meta-interpreter, new approach presented in this paper corresponds more to the meaning of the prefix meta.

We analyze the structure of expert systems (problem solvers) to specify a general description of a meta-interpreter. On that basis, we define the concept of an extendible meta-interpreter. The extendible meta-interpreter is divided into two parts – the kernel and its extension. While the kernel codes the functions that are common to most interpreters, the extension specifies the domain-specific functions of a particular interpreter.

1. Introduction

Meta-interpretation is one of the widely used programming techniques for writing rule-based expert systems. There are obvious reasons for extensive use of meta-interpreters: they are simple to use and understand, and, at the same time, they are very powerful. Many meta-interpreters have been written for special purposes ([16],[17],[19]). In this paper, we concentrate on meta-interpreters motivated by the construction of expert systems. We try to find a uniform paradigm for writing such a type of meta-interpreters which we call extendible meta-interpreters.

The extendible meta-interpreter consists of two parts, the kernel and its extension. The kernel gives a description of functions common to most meta-interpreters. Usually, it has an imperative character. On the other hand, domain-specific functions of a particular interpreter are encoded in the extension. The extension typically has a declarative character.

Hence, an extendible meta-interpreter is fully specified by the extension of its kernel. By this way, the emphasis is put more on the declarative style of programming. On the other hand, the hierarchical structure of the kernel helps in coding the imperative part of the meta-interpreter. Some examples of the PROLOG source code for the kernels and their extensions are given.

PROLOG is one of the languages that are widely used for writing rule-based expert systems and their inference machines. Thus in what follows, we use PROLOG to express our ideas about meta-intepreters, but our concepts are not confined to the logic programming paradigm.

The paper is organized as follows. In Section 2, the classical definition of a meta-interpreter is recalled and a well-known example of so called vanilla meta-interpreter is given. We use some other examples to motivate a different approach to meta-interpretation. This new concept of a meta-interpreter, which we call an extendible meta-interpreter, is outlined in Section 3. We give an informal definition of the extendible meta-interpreter in Section 4 and we also compare extendible meta-interpreters with classical meta-interpreters there. In Section 4.1, we define the extendible meta-interpreter and we describe its structure. We also present a road map to the structure of the extendible meta-interpeter there. We give some examples of extendible meta-interpreters written in PROLOG in Sections 4.2 and 4.3. We conclude with an overview of related and future research in Sections 5 and 6 respectively.

2. THE META-INTERPRETERS CLASSIC

Recall the classic definition of a meta-interpreter that is based on the following idea. Given an interpreter I of a programming language L, there are at least two ways how to write another interpreter I_1 of the same language L.

First, we can write I_1 as a completely new interpreter. This may be a complicated and a time consuming process, in particular, if there is only a little difference between the interpreter I and the interpreter I_1. In most cases, the process of developing I_1 is ineffective and the experience in developing the original interpreter I is of little help.

A better solution to the problem consists in writing a program P in L that changes the behaviour of the interpreter I to the behaviour of the interpreter I_1. The program P is called a meta-interpreter.

The above two ways to writing a new interpreter are shown on the Figure 2.1.

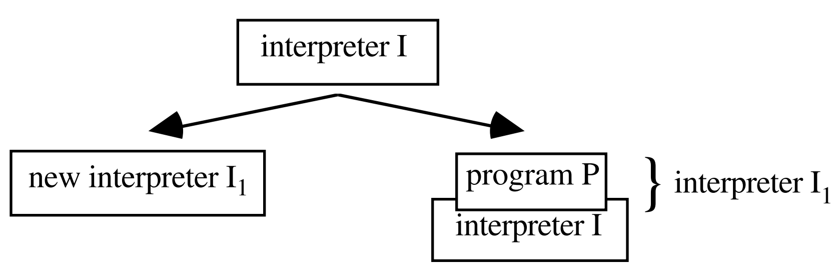

Figure 2.1 (two approaches to writing a new interpreter)

Hence, 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 1 [13]:

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

As you can see, the above definition is very simple. In fact, it is the relation between the language of the interpreter and the interpreted language what is essential for the definition of the meta-interpreter. For this reason, we speak about the syntactical basis of this definition. There is a philosophical problem related to the above definition: the definition which is rather syntactical by its nature, does not state what is an interpreter. More precisely, it does not say whether the interpreter is a program or a machine or both.

Comparing interpreters with meta-interpreters, we conclude that the interpreter is a program. It follows from the fact that, meta-intepreters represent a particular case of interpreters and according to the above definition, they are programs written in the interpreted language. Hence interpreters are programs, too.

We may still ask, however, where the machine was lost? Each program is simply a text, a code. It is like a word, and a word by itself cannot hurt, but saying a word can. To say a word we need a person, to interpret a program we need a machine.

When speaking about an interpreter, we assume that there is a machine that executes it. On the other hand, when speaking about an meta-interpreter, we assume that there is a machine with an appropriate interpreter that executes the program of the meta-interpreter (see the Figure 2.1).

Hence, there is a difference between the concepts of an interpreter and of an meta-interpreter. The classical definition does not take this difference into account and considers meta-interpreters and interpreters at the same level.

We recall two well-known examples of meta-interpreters now. They are written in PROLOG. As we already noted, PROLOG is a language suitable for writing meta-interpreters, and, more generally for writing meta-programs which are programs that use other programs as data. A meta-interpreter is a special case of a meta-program.

In PROLOG, we have the same structure of the program and the operated data. This is an obvious advantage for purposes of meta-programming. As the code of an 'object' program represents the data for its interpreter, the above feature of the language makes the task of writing a meta-interpreter easier.

However, this is not an exclusive property of PROLOG, it is well-known that LISP and some other languages have similar features, too.

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

Although Program 2.1 is a meta-interpreter according to the above definition, there is no visible advantage of using it instead of the standard PROLOG interpreter.

The following meta-interpreter is written on another level of abstraction which is usually called the *clause reduction level*. Meta-interpreters on that level make explicit the choice of clauses being used to reduce a goal, and the choice of a literal to generate the resolvent. Unification and backtracking are still handled implicitly, by the standard interpreter of PROLOG [16]. Most other meta-interpreters are derived by making extensions of this basic form. For this reason, the "basic" meta-interpreter (see below) is usually called the vanilla meta-interpreter in an obvious analogy with the ice cream flavors [12].

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

Program 2.2 (the vanilla meta-interpreter)

The clause reduction level mentioned above represents only one of many possible levels of abstraction on the computation of a meta-interpreter. When analysing the structure of meta-interpreters, one can identify various levels of abstraction. In what follows, the notion of an abstraction level will be very important and we shall give it a more precise description. For practical reasons, we shall identify the abstraction levels with granularity.

DEFINITION 2 [13]:

The *granularity* of a meta-interpreter corresponds to the level of access to the computation of the underlying interpreter. The higher the level of the access to the computation of the underlying interpreter, the finer the granularity of the meta-interpreter.

According to the above definition, the most rough granularity is attributed to meta-interpreters that use the computations of the underlying interpreter directly and with no visible access to any of its internal functions (like Program 2.1). The meta-interpreters that make visible the access to some of the functions of the underlying interpreter are of a finer granularity. Note that the vanilla meta-interpreter from Program 2.2 has a finer granularity than the simple meta-interpreter from the Program 2.1.

The concept of granularity is important for classifying meta-interpreters. The finer the granularity of a meta-interpreter, the more it may be able to change the computation of the underlying interpreter. In most cases, the fine granularity implies slower interpretation because the meta-interpreter itself has to be interpreted by an interpreter which we call a core interpreter of the programming language (see the Figure 2.1). Thus the key problem in choosing an appropriate granularity consists in finding a suitable compromise between the granularity of the meta-interpreter, that is implied by the requirements on changing the behaviour of the underlying interpreter, and the speed of its computation (i.e. interpretation).

It should also be noted that the meta-interpreters are doubling the space. We have already mentioned that a meta-interpreter is using two interpreters, a core interpreter of the language and a meta-interpreter to simulate another interpreter by changing the behaviour of the core interpreter. For this reason, many procedures may be doubled as they exist both in the core interpreter and in the meta-interpreter. It is obvious that this may cause a slow down of the computation. Although compilation usually solves problems of speed, note that the compiled meta-interpreter is not a meta-interpreter according to the above definition.

3. A New Approach To Meta-Interpretation

The following example (the Program 3.1) presents a program that simulates the computations of a finite automaton. It motivates a different view on meta-interpretation.

We call this program an interpreter because it interprets some code, in particular, the general description of an arbitrary finite automaton. It is not a meta-interpreter, according to the above definition, however, since there is a difference between the interpreted language, i.e., the description of the finite automaton, and the language of the interpreter.

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

Program 3.1 (the finite automaton simulator)

Note that the Program 3.1 has a similar structure as the vanilla meta-interpreter (Program 2.2). The possibility of making changes to the mechanism of the interpreter and making derivative programs is also saved.

The above examples demonstrate the main advantage of meta-interpreters consisting in an easy and simple access to the mechanism of the interpreter. One might be tempted to say that a meta-interpreter is an interpreter that gives an easy access to its own mechanism. Obviously, this is not a definition of a meta-interpreter since it does not say what means easy, but we think that it describes the character of meta-interpreters or a feature that users like on them more accurately. The core of that "definition" consists in the close relationship between the interpreter and the mechanism of this interpreter.

Our description of the concept of the extendible meta-interpreter will be closely related to the meaning of the prefix meta. This prefix is frequently used in constructions as a meta-program, a meta-theory, a meta-variable and it means something over the object level [1]: a meta-theory is a theory over another theory, a meta-variable is a variable that ranges over the (object) variables and a meta-interpreter is an interpreter of another interpreter. More precisely, the meta-interpreter is an interpreter that interprets a code, hence a description, of another interpreter. We shall explain this in more detail in Section 4.1.

In the following section we shall combine ideas of the above two pseudo-definitions into a compact definition of an extendible meta-interpreter.

4. The Structure Of The Extendible Meta-Interpreter

Let us compare our two pseudo-definitions.

A meta-interpreter is an interpreter which:

/a/ – enables an easy access to the mechanism of the interpreter, The pseudo-definition /a/ is a consequence of the definition of the meta-interpreter. A meta-interpreter written in the interpreted language enables in many cases an easy access to the mechanism of the interpreter. The definition /a/ is user-oriented, for users like everything what is easy. But we know that the meaning of the word easy was not explained in /a/, however.

The pseudo-definition /b/ is based on the meaning of the prefix meta. It could be a definition not only a pseudo-definition, if we defined what it is an interpreter and what is a description of an interpreter. However, there is still another problem with the pseudo-definition /b/: it depends on the interpreted program only, not on the interpreter itself.

It turns out that the role of an arbitrary interpreter is twofold:

- it is a meta-interpreter when it interprets some description of another interpreter and,
- it is only an interpreter if it interprets a program in the language which is not a description of an interpreter.

We shall show that there is a big gap between the pseudo-definition /b/ and the classical definition of the meta-interpreter (Definition 1).

Let us have any interpreter *I* of a given programming language *L*. A meta-interpreter of the same programming language *L* is a program *P* written in *L* which interprets the same codes (programs) as the interpreter *I*, hence 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 Programs 2.1 and 2.2, describing the simplest meta-interpreter and the vanilla meta-interpreter).

Moreover, the above interpreter *I* is a meta-interpreter, too. It follows from the pseudo-definition /b/, hence we shall call it a b-meta-interpreter if it interprets the above program *P* which gives a description of the interpreter.

Note that if we want to modify the mechanism of the b-meta-interpreter, we need not change the b-meta-interpreter *I*, but only the data, the code (i.e., the description) of the interpreter *P*. However, this is not a typical example of using the pseudo-definition /b/. It only shows that the pseudo-definition /b/ includes the classical approach, as well. It is well-known that changing data is easier than changing a program. Therefore, transforming the mechanism of the b-meta-interpreter should be easier.

The following table compares the concepts from the pseudo-definition /b/ with the classical approach.

	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/, in more detail in the following section where we try to bridge the gap by introducing a new concept of an extendible meta-interpreter.

4.1 Problem Solving and Meta-Interpreters

We shall show that similar result as above can be obtained by a different method. Namely, interprets a code of an interpreter. We will call the resulting meta-interpreter an extendible meta-interpreter. The following sequence of Figures describes a top-down construction of (the structure of) an extendible meta-interpreter.

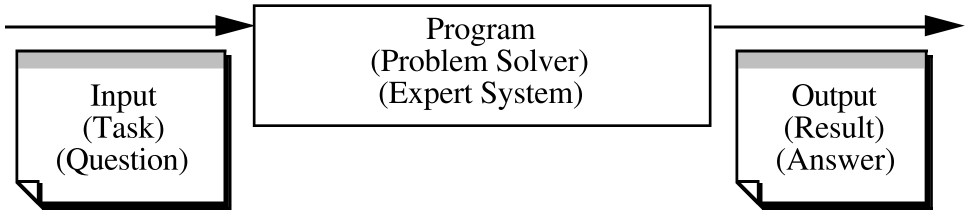

Figure 4.1 (program)

The Figure 4.1 shows the structure of an arbitrary program, or a problem solver or an expert system as a black box.

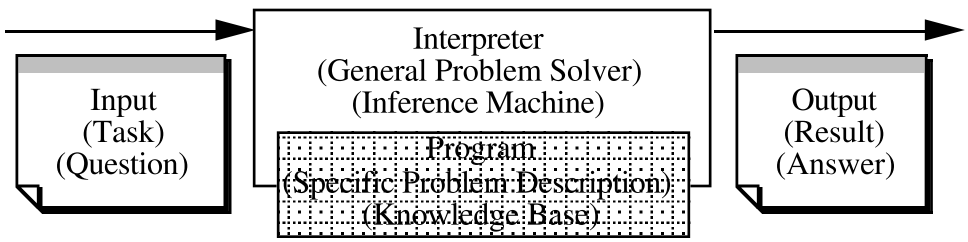

Figure 4.2 (the structure of the program)

The Figure 4.2 also shows the structure of an arbitrary program (a problem solver, an expert system) but in more detail. The program has two parts: an interpreter and a program description. Let us call the step from the description on the Figure 4.1 to the Figure 4.2 the *specification step* applied to a given program. The specification step describes more formally our previous discussion of using the prefix meta.

The Figure 4.2 corresponds to the current state in expert systems research. An empty expert system contains an inference machine and a knowledge base. A particular expert system can be defined by specifying a particular knowledge base. One does not change the inference machine but it is possible to specify the knowledge base.

However, sometimes we need to change the inference mechanism of the expert system. This can be obtained by performing the specification step to the interpreter.

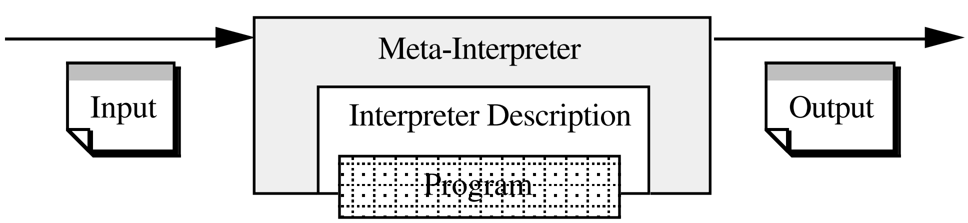

Figure 4.3 (new approach to meta-interpreters)

The Figure 4.3 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. We need easy and user friendly descriptions of the interpreter and of the inference machine respectively. It would be reasonable if both the program and the knowledge base could have an access to the mechanisms of the interpreter and of the inference machine through their descriptions.

The Figure 4.3 corresponds to the pseudo-definition /b/. Of course, we could now continue in performing further specification steps which would result in a meta-meta-interpreter with the description of the meta-interpreter, etc. There is no visible gain of doing it, however.

Now, we shall discuss some problems concerning terminology. We are trying to follow two goals: we would like to have a terminology consistent with the classical approach to meta-interpretation and, at the same time, we would like to generalize the concept of the meta-interpreter. We should note a difference between the classical approach and that one adopted here. The classical approach is based on specialization while the approach adopted here uses generalization. The difference between the concepts of these two approaches is shown on the Figure 4.4. We think that the latter approach is more general, because generalization includes specification, as shown in the discussion at the beginning of the Section 4.

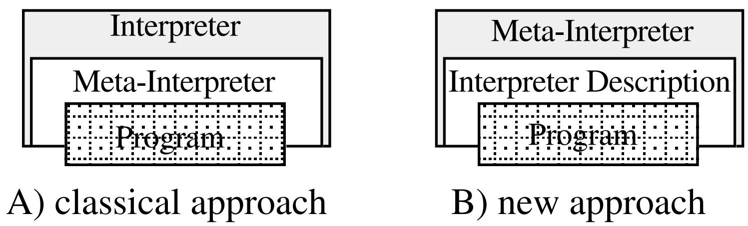

Figure 4.4 (comparison of the approaches to meta-interpretation)

There is a simple solution to the above terminology problem which is based on renaming the concepts of the new approach:

the meta-interpreter	→	the kernel of the extendible meta-interpreter
the description of the interpreter	→	the extension of the kernel.

We can now describe an extendible meta-interpreter by simple equation:

extendible meta-interpreter = kernel + extension.

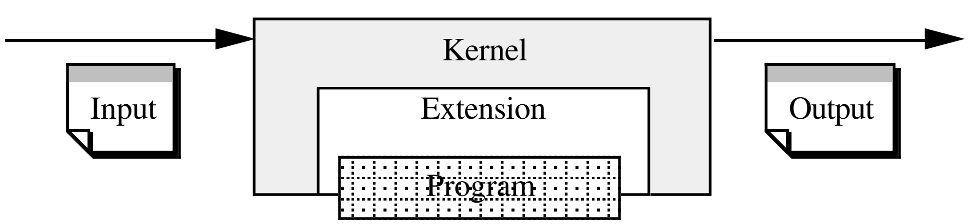

Figure 4.5 (the structure of an extendible meta-interpreter)

The Figure 4.5 displays the structure of an extendible meta-interpreter. We may ask whether it is possible to write a fixed kernel and complete it by various extensions to various interpreters. We shall explain roles of all parts shown in the Figure 4.5. There is a program describing an algorithm to solve a particular problem. It is written in a programming language *L*. The extension of the kernel represents a description of a particular interpreter, e.g., in (program) for interpreting programs and the kernel and its extension put together a particular interpreter of the programming language in question.

Now, we shall compare extendible meta-interpreters to classical meta-interpreters. First, the structure of the extendible meta-interpreter is scalable. The Figure 4.6 explains what does it mean.

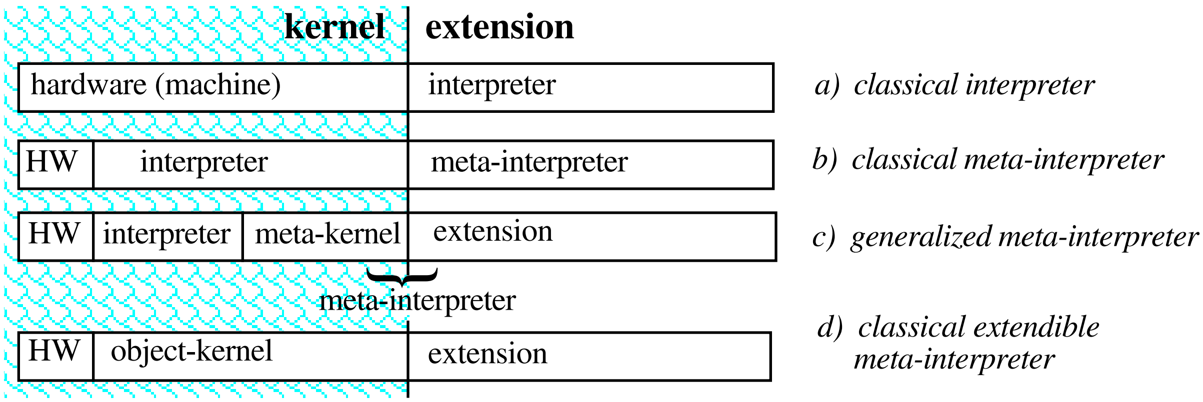

Figure 4.6 (scalable structure of the extendible meta-interpreter)

Note that the parts a) and b) represent current approach to interpretation. We discussed above some problems of classical meta-interpreters, in particular we noted that it is difficult to change their behaviour (part a) and that they are less efficient and are doubling the space (part b). Extendible meta-interpreters from the parts a) and b) and meta-interpreters have one feature in common: the extension has an imperative character and therefore it is difficult to change the extendible meta-interpreter. The part b) also shows that the classical approach to meta-interpretation can be expressed in terms of the extendible meta-interpreter.

The parts c) and d) are more interesting. We shall discuss the part c) later in this paper (Section 4.2), as it is similar to a classical meta-interpreter. Therefore it inherits some less favourable features of meta-interpreters, in particular inefficiency and doubling memory space.

The part d) represents the structure of extendible meta-interpreters 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 extendible meta-interpreter (the part d) of the Figure 4.6) 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 give examples of extendible meta-interpreters with the structure similar to the part c) of the Figure 4.6 which show that it is easy to write the meta-kernel and the extension in a high-level language. By compiling the meta-kernel and linking with an interpreter 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 general definition of an extendible meta-interpreter.

DEFINITION 3:

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

The power of an extendible meta-interpreter is obtained by the separation of the kernel and its extension. Features which are common to various interpreters can be hidden in the kernel, while the extension contains domain-specific information for a particular interpreter, the extension being encoded either in the interpreted language or in its superset. Such an organization of the extendible meta-interpreter gives an easy access to the mechanism of the interpreter.

The definition of an extendible meta-interpreter covers all parts of the Figure 4.6. For example, a LISP machine (the kernel) and a LISP interpreter (the extension) written in LISP make an extendible meta-interpreter. Its structure corresponds to the part a) of the Figure 4.6. All PROLOG meta-interpreters make the extensions of extendible PROLOG meta-interpreters. Nevertheless, these two examples are not typical for extendible meta-interpreters because of the imperative character of the extension.

We prefer the declarative style of programming of the extension because it is easier to write a declarative description of the interpreter than an imperative one. For the same reason, we also prefer meta-interpreters consisting of the meta-kernel in an imperative style and the extension in a declarative style (see the part c) of the Figure 4.6). We call an extendible meta-interpreter an *easily extendible meta-interpreter* if it has a declarative extension.

In what follows, we shall discuss only easily extendible meta-interpreters. They have the structure shown in the part c) of the Figure 4.6, where the kernel consists of a machine, an interpreter and a meta-kernel. In this case, we shall call the meta-kernel simply the 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 EXTENDIBLE META-INTERPRETERS IN PROLOG

In this section we shall present some examples of easily extendible meta-interpreters in PROLOG. We shall show some PROLOG kernels (meta-kernels) and their extensions. So far, we have used examples of meta-interpreters, that were mostly derivatives of the vanilla meta-interpreter. We did so to find out what these meta-interpreters have in common, to grasp these common features and encode them in the kernel.

The simplest feature that can be identified in many meta-intepreters consists in using the predicate `solve` in programming meta-interpreters or interpreters. We shall call this feature a *Zero Level Kernel*, since it implies an empty meta-kernel and it corresponds to the case b) from the Figure 4.6. The Zero Level Kernel is specified by the following predicate:

```
solve(Goal)
```

or more generally by the binary predicate

```
solve(Goal,Result).
```

The arguments of the predicate `solve` of the meta-interpreters can be divided into two groups: input (goal) and output (result) arguments. It follows that, almost every current meta-interpreter is an extension of the Zero Level Kernel and we can say that the Zero Level Kernel corresponds to current state of art in programming meta-interpreters.

Obviously, the Zero Level Kernel is not very interesting from the point of view of extendible meta-interpreters. We shall discuss now a more powerful kernel which we call the *Half Level Kernel*. It is non-empty and reflects the fact that every goal can be solved in three possible ways. First, primitive (empty) goals are solved in one step (for example `true` in PROLOG). Second, more complex goals are transformed to other (preferably simpler) goals. Third, some goals have no solutions. The Half Level Kernel consists of a (PROLOG) program that implements these three ways.

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

Program 4.1 (the Half Level Kernel)

The Half Level Kernel is appropriate for implementing some simple search algorithms. We shall show the extension of the Half Level Kernel which describes the depth-first search. The resulting program (the Program 4.1 and the Program 4.2) can be seen as an extendible meta-interpreter that interprets a program consisting of the description of a particular graph (the edges) and a set of final nodes.

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

Program 4.2 (the extension for search)

We can also write an extension of the Half Level Kernel implementing a simple PROLOG interpreter corresponding to the vanilla meta-interpreter[1].

```
empty_goal(true,yes).

transform_task((A,B),NewGoal,not_used):-
        transform_task(A,NewA,_),
        and(NewA,B,NewGoal).

transform_task(A,B,not_used):-
        A\=true,A\=(_,_),
        clause(A,B).

customize_solution(not_used,yes,yes).

rest_solution(_,no).
```

Program 4.3 (the extension for the interpreter of PROLOG)

In the above extension as well as in many other meta-interpreters, the process of transforming the goal consists of three steps: selection of a subgoal, expansion or reduction of this subgoal, and, finally, making a new goal. This simple idea is reflected in the *First Level Kernel*.

```
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)

Some new notions appear in the Program 4.4, namely the Frontier (also used in the Half Level Kernel) and the Rule. The variable Frontier contains information about the choice of the selected atom from the goal, which is used in the process of constructing a new goal. Sometimes, we need to add some additional information to the Frontier in the process of constructing a new goal. This information is used later when customizing the solution.

The variable Rule contains information about the transformed goal, for example the description of the rule (the clause) used in the transformation. This information is used when customizing the solution, too.

Now, it is not difficult to write an extension of the First Level Kernel by implementing a simple PROLOG interpreter. In fact, it will have the same power as the extension of the Half Level Kernel described by the Program 4.3.

```
empty_goal(true,yes).
select_subgoal((A,B),Goal,[B|Rest]):-
      select_subgoal(A,Goal,Rest).
select_subgoal(Goal,Goal,[]):-
      Goal\=true,Goal\=(_,_).
expand_goal(Goal,ExpandedGoal,not_used):-
      clause(Goal,ExpandedGoal).
make_task([B|Rest],ExpandedGoal,NewTask):-
      make_task(Rest,ExpandedGoal,A),
      and(A,B,NewTask).
make_task([],Goal,Goal).
customize_solution(_,_,yes,yes).
rest_solution(Task,no).
```

Program 4.5 (the extension for the interpreter of PROLOG)

Note that the higher is the level of the kernel, the more complex structure has its extension. This is an obvious consequence of the increasing power of higher level kernels and of their finer granularity. The hierarchy of kernels is partially based on their history: we started with the Zero and the First Level Kernels, because we mostly worked with meta-interpreters of PROLOG. Then we added the simplified version of the First Level Kernel which we called the Half Level Kernel. We did it for the sake of implementing search. We have used the term Half Level, because we wanted to preserve the hierarchy of levels. We stopped at the Second Level Kernel. Note that adding the higher level kernels is also possible. It should be said, however, that kernels of levels higher than two become increasingly dependent on the interpreted language.

Now, we shall say a word about the interface between the kernel and its extension. In the above examples, the interface consists of the list of predicates. It is due to our choice of PROLOG as the language for writing extensions. The user defined predicates make hooks in the kernel where the user can hang the procedures that modify the behaviour of the kernel. Then the corresponding extension consists of definitions of (the programs for) these predicates.

We shall consider the interfaces to the above mentioned kernels. Obviously, the interface to the Zero Level Kernel is the simplest. It consists of the only predicate

`solve`.

We have already noted that for this reason, almost every PROLOG meta-interpreter is an extension of the Zero Level Kernel. The interface to the Half Level Kernel is more complicated. It consists of the predicates

`empty_goal`, `transform_task`, `customize_solution` and `rest_solution`.

The interface to the First Level Kernel is still more complicated. It consists of the following predicates:

`empty_goal`, `select_subgoal`, `expand_goal`, `make_task`, `customize_solution`

and

`rest_solution`.

4.3 OTHER EXTENSIONS AND THE SECOND LEVEL KERNEL

We start this section with a more complicated example of extension of the First Level Kernel. This extension (the Program 4.6) and the First Level Kernel (the Program 4.4) fully describe an extendible PROLOG meta-interpreter that computes proofs[2]. We also use this extension to present possible structure of the Frontier. This extension has the following property: it requires some additional information when a new task is created and this information is used again later in the process of customizing solution. Note that the First Level Kernel (the Program 4.4) and this extension (the Program 4.6) make the core of the above mentioned 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,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=collapsed,Ch=not_collapsed).
make_task([],Goal,Goal).

customize_solution([(_,not_collapsed)|T],Rule,(SProofA,ProofB),
                   (ProofA,ProofB)):-
            customize_solution(T,Rule,SProofA,ProofA).
customize_solution([(_,collapsed)|T],Rule,ProofB,(ProofA,ProofB)):-
            ProofB\=failed,
            customize_solution(T,Rule,fact,ProofA).
customize_solution([],A-B,ProofB,A-ProofB):-
            ProofB\=failed.

rest_solution(_,failed).
```

Program 4.6 (the extension for the interpreter of PROLOG with proofs)

We can also use the same kernel (the Program 4.4) for writing a completely different extendible meta-interpreter. The only thing we have to do is to write a new extension. The following program is an example of using the First Level Kernel for a quite different interpreter. The program is similar to the Program 3.1: it simulates a finite automaton.

```
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 4.7 (the extension for the finite automaton with proofs)

If we do not include the predicates of the interface that are not used in the bodies of the clauses the interpreted language consists of only two predicates, namely, `final_state` and `rule`. This approach corresponds to the standard 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 4.6 and 4.7, the later is rather declarative while the former is imperative. Therefore the First Level Kernel is satisfactory for the simulator of a finite automaton, but it is not suitable for the extendible PROLOG meta-interpreter (the extension should have a declarative character). We introduce the *Second Level Kernel* to save the declarative character of the extension.

Note that, there are three predicates of imperative character in the Program 4.6, 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, namely as a list of pairs. This list (the Frontier) arose from a process of subgoal selection where the first components of the pairs of the list were instantiated while the second components remainded free (see the Programs 4.6 and 4.8). The second components of the pairs can be instantiated in the process of making a new task where the first components are already used (see again the Program 4.6). Finally, the Frontier can be used in customizing the solution (like in the Program 4.6). Note that, the length of the Frontier is determined by the "depth" of the task which is equal to the number of steps which are used to find a subgoal of this task. By this way, the depth of the task is determined by the structure of the task and by the strategy of subgoal selection.

We introduce here new concepts, namely, the *meta task* and the *simple task*. One can select directly 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 decomposed into some "independent" subtasks using the predicate `goal_selection`. By this way, the structure of the meta task remains 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 extendible PROLOG meta-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 4.8 (part of the Second Level Kernel)

The Second Level Kernel consists of the First Level Kernel (the Program 4.4) and of the Program 4.8. 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 fully determines the processes of making a new task and customizing solution."

PROLOG is an example of language that satisfies this criterion. In most cases, we do not need customize the solution explicitly. However, the languages that do not satisfy the above criterion can be interpreted by the Second Level Kernel as well, but they are not supported i.e. the programmer has to code all the user defined predicates even if the extendible meta-interpreter will not use them all.

The Second Level Kernel can be used for a wide range of extendible meta-interpreters. It is easy to prove that every extendible meta-interpreter, that can be written with the use of the Second Level Kernel, can also be written with the use of 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 would then represent a shell of the inference machine.

5. Related Research

In this section, we shall compare the idea of extendible meta-interpreters with a similar approach based on skeletons. The theory of skeletons is a significant part of a methodology for systematically building complicated Prolog programs from standard components [15].

Skeletons are basic Prolog programs with a well-understood control flow. Applying a *technique*, a standard Prolog programming practice, to a skeleton creates an *extension* of the skeleton. It is possible to create different extensions of the same skeleton, each for a specific feature of the desired program. Finally, separate extensions of the same skeleton can be automatically or semi-automatically composed into a single program. An example of using skeletons in development of a Prolog tracer can be found in [8].

At the beginning, we should note that skeletons are used for a slightly different purpose than the extendible meta-intepreters are. While the skeletons are primary dedicated to simplifying the process of complicated Prolog program development, extendible meta-interpreters are more oriented to the area of meta-interpretation and interpretation in general. However, the idea of the extendible meta-interpreter, hence of dividing the program into the kernel and its extension, can be also used in the development and maintenance of complicated programs.

The following difference between the skeleton and the kernel is more serious. While the skeleton is a program, a stand-alone application, and the extension of the skeleton is also a stand-alone application, the kernel and its extension are just modules of an extendible meta-interpreter. Therefore, to develop an extendible meta-interpreter one needs both the kernel and the extension of the kernel. Finally, programming an extension of the skeleton includes changing the skeleton, i.e., changing the existing code, while programming the extension of the kernel is nothing else than adding a new code to the kernel. The only thing one has to follow is the structure of the interface between the kernel and its extension. In our opinion, this difference implies that the development of the extension of the skeleton is more complicated than the development of the extension of the kernel.

Despite the above mentioned differences, skeletons and extendible meta-intepreters are closely related to each other. Identifying an appropriate skeleton is similar to finding a kernel, although programming the kernel could be a little bit complicated process because the programmer has also to design an appropriate interface between the kernel and its extension. But this effort is definitely paid off by easier development of future extensions of the kernel. Since separate extensions of the same kernel are based on the same set of interface predicates, it is also easier to compose them into a single extension which includes features of parent extensions.

6. FUTURE RESEARCH

As we sketched above, the idea of extendible meta-interpreters can be used in the development and maintenance of complicated programs. However, contrary to the skeletons, the methods for automatic or semi-automatic composition of various extensions have not been drawn up yet for extendible meta-interpreters. So, it is the first open area of conceivable research.

The second area of interest is using techniques of extendible meta-interpreters in the construction of HCLP (Hierarchical Constraint Logic Programming) interpreters [18]. Inspired by the meta-terms and attributed variables [9], we suggest to use extendible meta-interpreters in a similar manner [5]. While the meta-terms generalize the process of unification and they are suitable for implementing CLP interpreters therefore, extendible meta-interpreters generalize the whole process of interpretation including unification. So, extendible meta-interpreters could serve as a platform for implementing various HCLP interpreters with inter-hierarchy comparison. This also fulfills our original goal of using extendible meta-interpreters for expert systems construction because, in our opinion, HCLP with inter-hierarchy comparison is suitable for expert systems shell composition [4].

7. CONCLUSIONS

In this paper, we have described a new approach to meta-interpretation, based on the concept of an extendible meta-interpreter. The extendible meta-interpreter preserves the positive features of meta-interpreters, namely, an easy access to the mechanism of the interpreter and, at the same time, the possibility to suppress the slow down of the computation and doubling the memory space.

The idea of an extendible meta-interpreter is based on the separation of the general part of an interpreter from the domain-specific one. The extendible meta-interpreter consists of two parts: the kernel and its extension. The hierarchical structure of the kernel, makes it possible for the user to 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 (meta)interpreters.

A uniform frame for writing (meta-)interpreters helps the programmer to concentrate on features of a particular (meta-)interpreter without troubles with general principles of interpretation. The idea of the extendible meta-interpreter can also help as a consolidating element in the reflective programming. Our approach can help in composing interpreters or developing program modulants, enhancements and mutants [16], too.

We have concentrated mostly on using extendible meta-interpreters as a tool for the construction of inference machines of expert systems and problem solvers. An example of the extendible meta-interpreter for search, a classical technique used for construction of expert systems, has been given. The concept of an extendible meta-interpreter was originally motivated by the research into meta-interpreters for building expert systems. Using an extendible meta-interpreter simplifies and speeds up the construction both of a particular inference machine and of a particular expert system.

Throughout the paper, we used PROLOG to demonstrate some examples of meta-programs. This does not imply that the results of the paper are confined to the logic programming paradigm. The above results can be applied to other programming environments, too. We have also presented some example programs to show that idea of extendible meta-interpreters is practical and useful.

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] Barták, R. and Štěpánek, P., Meta-Interpreters and Expert Systems, Tech. Report No 115, Department of Computer Science, Charles University, October 1995

[4] Barták, R., Expert Systems Based on Constraints (in Czech), Doctoral Dissertation, Charles University, Prague, 1997

[5] Barták, R., A Plug-In Architecture of Constraint Hierarchy Solvers, to appear in *Proceedings of PACT'97*, London, April 1997

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

[7] Jain, A., Sterling, L. and Kirschenbaum, M., Towards Reusability Based Upon Similar Computational Behaviour, in: *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, Rockville, Maryland, USA, June 1995

[8] Lakhotia, A., Sterling, L. and Bojantchev, D., Development of a Prolog Tracer by Stepwise Enhancement, in: *Proceedings of the Third International Conference on Practical Applications of Prolog*, Paris, April 1995

[9] Meier, M., Brisset, P., Open Architecture for CLP, TR ECRC-95-10, ECRC, 1995

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

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

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

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

[14] Sterling, L., Jain, A. and Kirschenbaum, M., Composition Based on Skeletons and Techniques, Work presented at ILPS `93 Post Conference Workshop on Methodologies for Composing Logic Programs

[15] Sterling, L. and Kirschenbaum, M., Applying Techniques to Skeletons, in: *Constructing Logic Programs*, J.M.J. Jacquet (editor), John Wiley & Sons, 1993

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

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

[18] Wilson, M., Borning, A., Hierarchical Constraint Logic Programming, TR 93-01-02a, Department of Computer Science and Engineering, University of Washington, May 1993

[19] Yalçinalp, L.Ü. and Sterling, L., An Integrated Interpreter for Explaining PROLOG's Successes and Failures, Case Western Reserve University, CES TR-88-04, April 1988