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 classical 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.

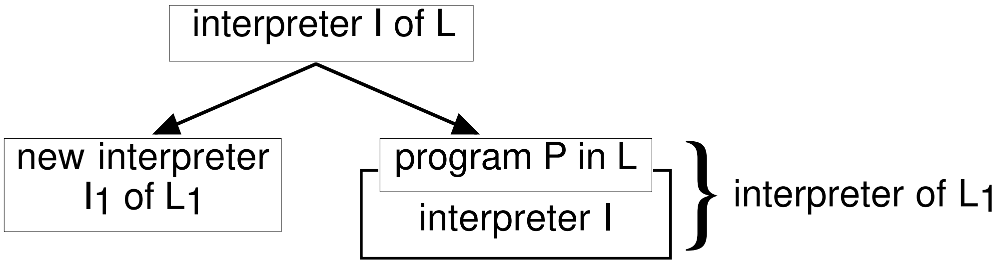
We analyze the structure of expert systems (problem solvers) to specify a general description of a meta-interpreter. On that basis, we define the concept of a generalized meta-interpreter which we call a mega-interpreter. The mega-interpreter is divided into two parts – the kernel and its extension.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

META-INTERPRETERS CLASSIC

DEFINITION [10]:
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.

- Why are meta-interpreters so popular?

  — modify behaviour of underlying (core) interpreters



  — easy access to the mechanism of the interpreter

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

  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).

- Why do we want to enhance meta-interpreters?

  — finer granularity means slower interpretation

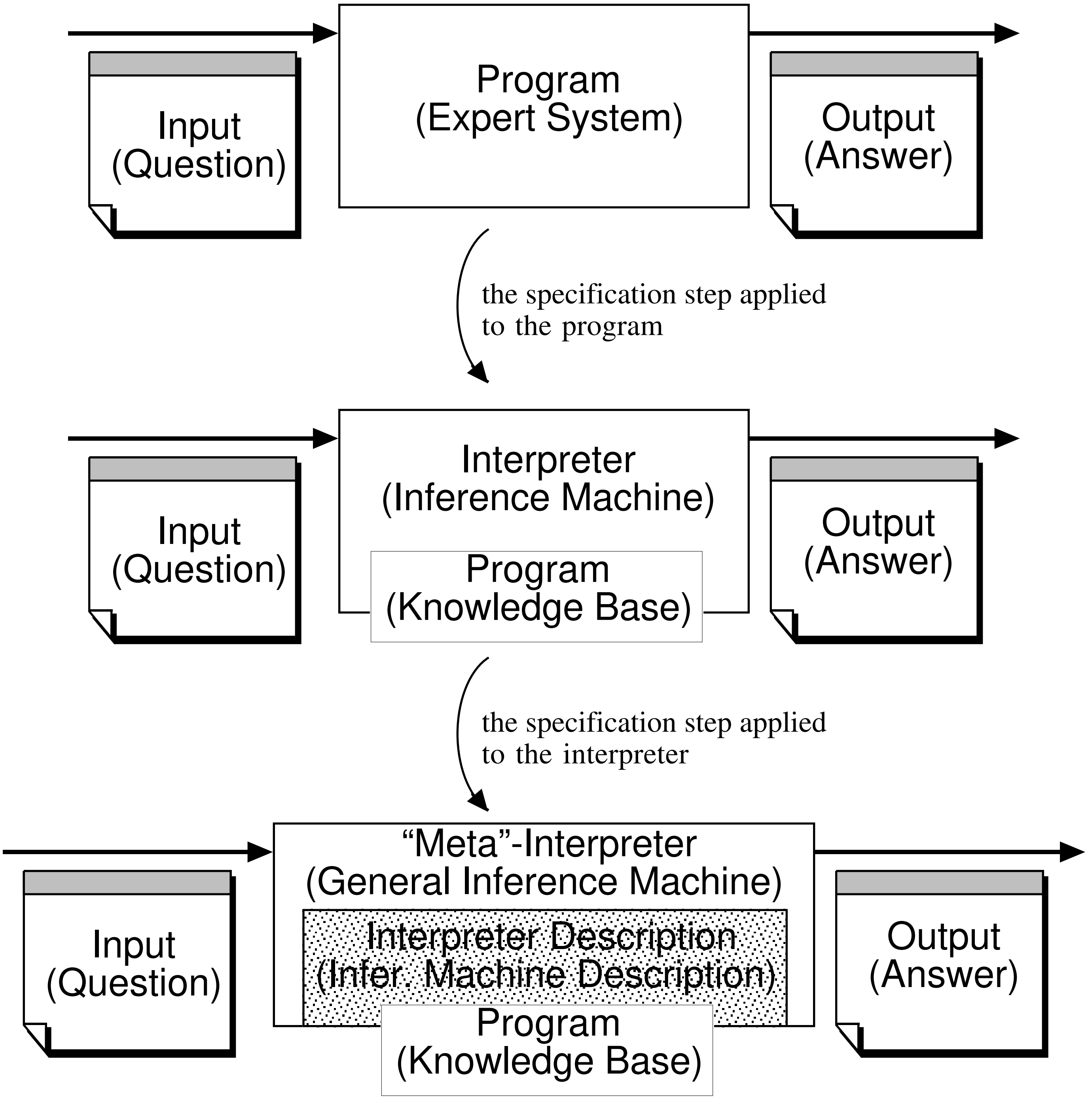
  The meta-interpreter itself has to be interpreted by an interpreter which we call a core interpreter of the programming language.

  — space doubling

  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 the problems of speed, note that the compiled meta-interpreter is not a meta-interpreter according to the above definition.

ANALYSING INTERPRETERS

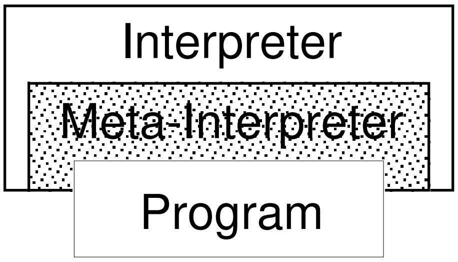
Our description of the concept of the mega-interpreter will be closely related to the meaning of the prefix meta. This prefix is frequently used in constructions as a meta-program, a meta-theory, a meta-variable and it means something over the object level [1]: a meta-theory is a theory over another 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.



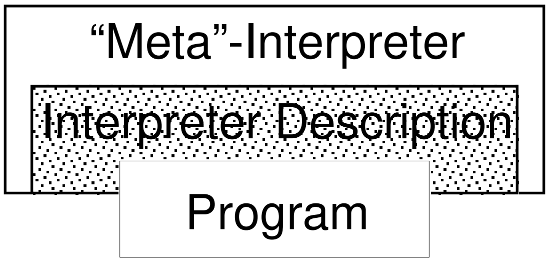
Of course, we could now continue in performing further specification steps which would result in a meta-meta-interpreter with the description of the meta-interpreter, etc. There is no visible gain of doing it, however.

INTRODUCING MEGA-INTERPRETERS

We should note a difference between the classical approach and that one adopted here.



classical approach



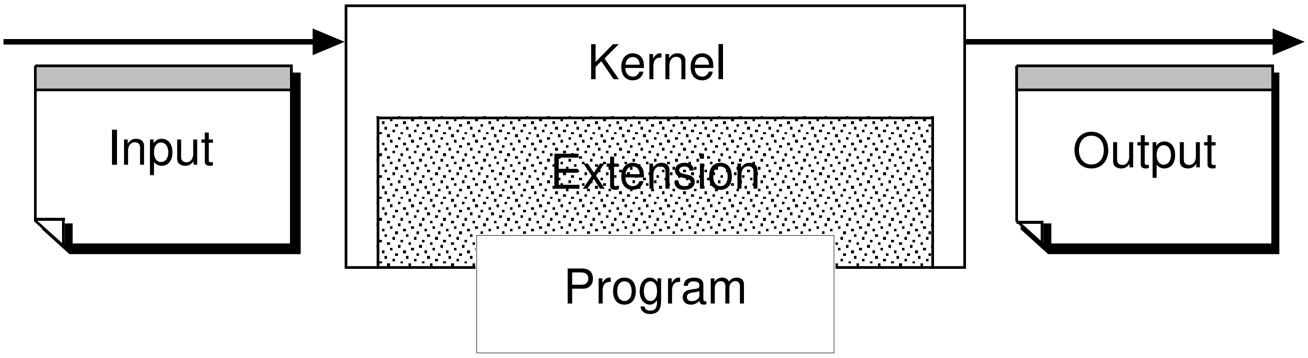
new approach

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. So…

"meta"-interpreter → the kernel of the mega-interpreter

interpreter description → the extension of the kernel

MEGA-INTERPRETER = Kernel+Extension

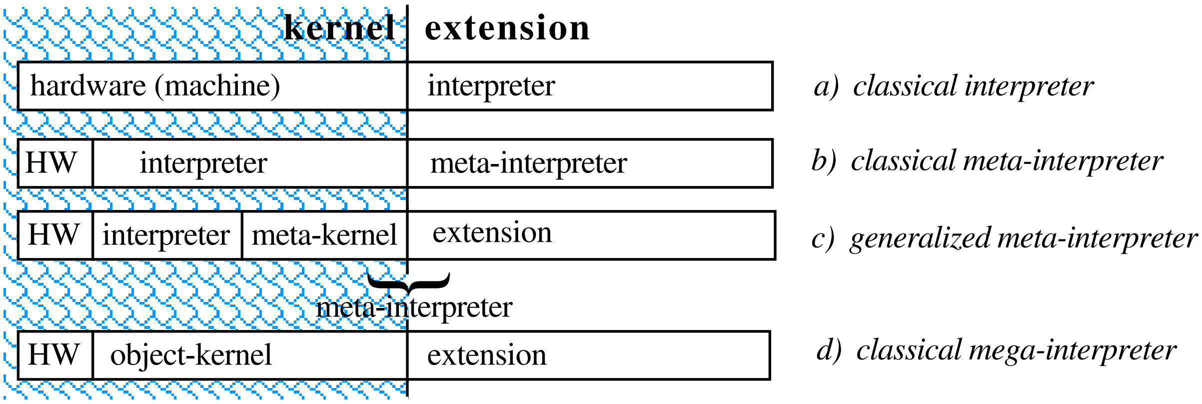


DEFINITION:

A *mega-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.

Mega-Interpreters In Detail

- scalable structure



Note that 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).
The part c) is similar to a classical meta-interpreter hence it inherits some less favourable features of meta-interpreters, in particular inefficiency and doubling memory space.
The part d) represents the structure of mega-interpreters with the kernel written in the object (machine-oriented) language and the extension written in the interpreted high-level language. We can obtain the object-kernel by compiling the meta-kernel and linking it with the interpreter. The classical mega-interpreter 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.

- power

The power of a mega-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 mega-interpreter gives an easy access to the mechanism of the interpreter.

- easy of use

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). We call a mega-interpreter an easy mega-interpreter if it has a declarative extension.

MEGA-INTERPRETERS IN PROLOG

- The Half Level Kernel

```
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).
```

— the extension for PROLOG interpreter

```
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).
```

- 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).
```

VARIOUS EXTENSIONS

• The Extension for Search

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 (extension+kernel) can be seen as a mega-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).
```

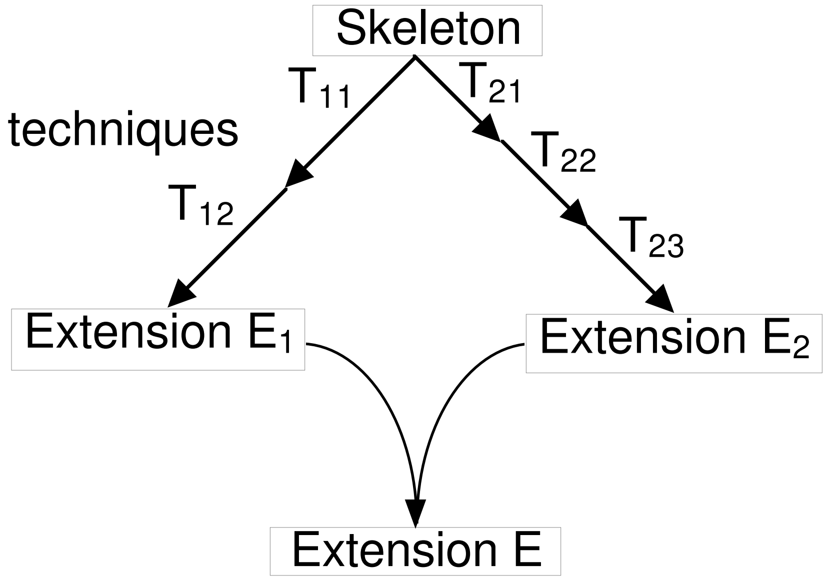
• The Extension for the Finite Automaton with Proofs

We can also use the same kernel (the Half Level Kernel) for writing a completely different mega-interpreter. The only thing we have to do is to write a new extension. The following program simulates a finite automaton and returns the path of computation. Notice the usage of the Frontier.

```
empty_goal([]-Q,accept):-
        final_state(Q).
transform_task([A|T]-Q,T-NewQ,Q-NewQ):-
        rule(Q,A,NewQ).
customize_solution(Q-NewQ,S,NewQ-S):-
        S\=no.
rest_solution(_,no).
```

Related Research

The theory of skeletons is a significant part of a methodology for systematically building complicated Prolog programs from standard components [12].

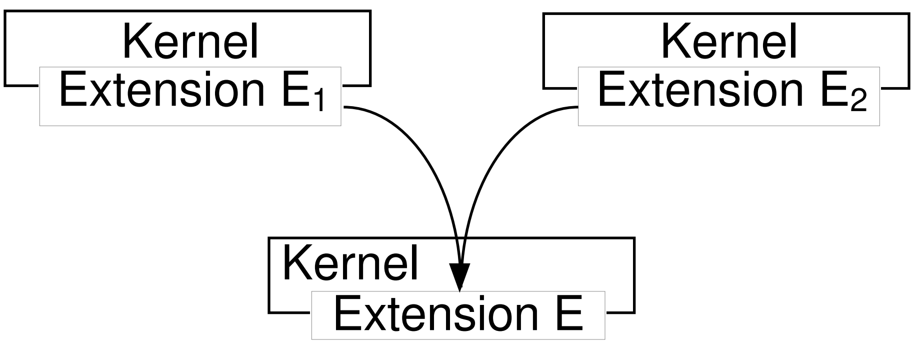


Skeletons are basic Prolog programs with a well-understood control flow.

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 automaticaly or semi-automaticaly composed into a single program.

While the skeletons are primary dedicated to simplifying the process of complicated Prolog program development, mega-interpreters are more oriented to the area of meta-interpreters and interpreters in general. Hovewer, the idea of the mega-interpreter, hence of dividing the program into the kernel and its extension, can be also used in the development and maintenance of complicated programs.



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.

Since separate extensions of the same kernel are based on the same set of interface predicates, we assume it is easier to compose them into a single extension which includes features of parent extensions.

Future Research

- Program Development

  development of methods for automatic or semi-automatic composition of various extensions into one extension

- Concurrent Extensions

  subsumption support for concurrent extensions to one kernel and switching between them

- The Kernel as a Stand-Alone Application

  development of a small default extension that could be included into the kernel

- Mega-Interpreters and Expert Systems

  applying techniques of mega-interpreters to the construction of inference machines of expert systems

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

[5] 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

[6] 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

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

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

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

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

[11] 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

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

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

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

[15] 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