A MULTILEVEL KNOWLEDGE REPRESENTATION OF STRATEGIES FOR COMBINING MODULES [a]

M. BIELIKOVÁ, P. NÁVRAT
Slovak University of Technology, Dept. of Computer Science and Engineering, Ilkovičova 3, 812 19 Bratislava, Slovakia
{bielik,navrat}@elf.stuba.sk
http://www.dcs.elf.stuba.sk/~bielik, http://www.elf.stuba.sk/~navrat

The research was motivated by the need to represent knowledge (when written in Prolog) in a way that would retain its inherent structure. Our approach attempts to offer means of abstraction for structuring logic programs according to both generality levels and to knowledge content, i.e. meta-levels. Using the notion of a modular logic program, we define a special reflection mechanism which establishes connections between modules at a given level and a level above. We describe various ways of combining of modules. We propose defining meta-knowledge in separate modules, with each module defining one particular strategy. Another option is dividing meta-knowledge that defines one strategy into several modules at one level, and defining a way of combining them.

1 Problem Area and Goal

Despite several widely recognized advantages of logic programming such as declarative semantics, or mechanisms of unification and deduction, the extent of its suitability for development of large software systems is relatively limited. One of the crucial problems is the lack of concepts, and consequently of language constructs, for structuring programs, modularity, sharing and hiding, etc. All of them are important means to manage complexity of software.

The need to structure logic programs is motivated also by another perspective. Logic programming is a paradigm which is suitable to writing programs that embody the relevant problem knowledge written in a form that is close to the one in which it is expressed by human experts. However, the knowledge pieces are frequently related to each other according to various properties: some of them can be more general than some others, some can define ways how to evaluate, select, or manipulate other ones, etc. It would be very useful to keep such relations represented in the knowledge base. Less general knowledge pieces could inherit some of the properties of the more general ones without having to re-represent them. Knowledge pieces that manipulate other pieces could be grouped to form an explicit meta-level of knowledge representation.

The desire to have in a logic programming language such as Prolog means

[a] The work reported here was partially supported by Slovak Science Grant Agency, grant No. 95/5195/605.

for dividing a software system into smaller relatively separated and independent units with transparent minimal interfaces has been responded by several authors. Separated logic databases are called modules,[6,1,10,24,25,16,12] theories,[15] units.[13] Several authors have applied concepts of the object oriented programming to achieve structuring of logic programs.[11,28,8,14,17,15,23,27]

Problems are encountered when trying to combine logic databases (modules). Several approaches have been tried, e.g. inheritance,[17] context switching,[13] introducing implication into goals,[19,9] different definitions of visibility of atoms,[15,25] using abstraction in separating the logic database from the concrete implementation by specifying required resources and produced results.[24]

Mutual communication among logic databases has not been solved satisfactorily so far. No generally applicable strategy has been proposed that could be used in developing an arbitrary system. Moreover, it seems that domain dependent knowledge plays an important role in deciding on what is the suitable way of combining logic databases for a given problem.

The above described difficulty is often approached with the technique of meta-programming. A straightforward usual way of using meta-programming in logic is based on defining a meta-interpreter that defines explicitly every logic program processing machine instruction, taking into account the chosen level of meta-interpret's granularity.[26]

An alternative approach is to allow direct access to some specific parts of the abstract program processing machine's state. The technique is called introspection, or reflection. As a consequence, it is not necessary to model at the meta-level the whole computing process, but only those its parts which are to be modified. The approach is not entirely new,[7] but not so much attention has been paid to it as to meta-interpretation. Using introspection in logic programming allows explicit representation of knowledge about communication among logic databases at the meta-level, while the solution of the problem is defined at the object (program) level.

We shall present our proposal how to extend a logic program with a possibility of representing parts of it at several levels. Also, we propose how to use such a multilevel logic program to implement various ways of combining logic databases. Our approach is based on the reflection technique as elaborated by Lamma, Mello, and Natali[13] who used reflection for combining Prolog databases through contexts and inheritance.

2 Reflection and Multilevel Logic Programs

Rather than (meta-) interpreting the overall behaviour of an abstract machine, some parts of the machine's state are made available to be accessed and manipulated directly through the reflection mechanism. The reflection mechanism switches the computation from the (object-) level to the introspective (meta-) level domain (upward reflection) and vice versa (downward reflection).[13] The object level machine's visible state ought to be chosen to suit needs of the problem domain. Let us assume the visible state is the triplet $(M, Goal, AUX)$, where M is the current module, $Goal$ is the current (sub-)goal, and AUX is a term representing auxiliary information. This is one particular choice of the level of abstraction for the reflective operations.

Both the levels of a program are represented in the same way – as modules. Connection between an object level module M and a meta-level module $Meta$ is defined by the relation *connect*. If *connect(Meta)* can be proved in module M then module $Meta$ is a meta-module of M.

Now, let us assume the computation takes place at the object level in a module M. When a module $Meta$ becomes a meta-level module of M by proving the goal $connect(Meta)$, there is provided an implicit upward reflection. An attempt to prove $reflect\_up([M, Goal, AUX])$ shifts the computation to the meta-level where the visible state of the abstract machine is explicitly available through the triplet. The attempt can either succeed or fail. If it fails, the failure is reported in the object level in the usual way.

When the computation takes place at a meta-level, an explicit downward reflection is attempted by the goal $reflect\_down([M1, Goal1, AUX1])$ (see Figure 1). This causes an object level computation to start in the module $M1$ aiming to prove the goal $Goal1$. Again, the attempt can either succeed or fail, similarly to the above case. If it succeeds, new visible state is reflected up by an implicit upward reflection. In such a way, results of the object level computation become available at the meta-level. If it fails, the failure is reported to $reflect\_up$ goal at the meta-level which fails, too.

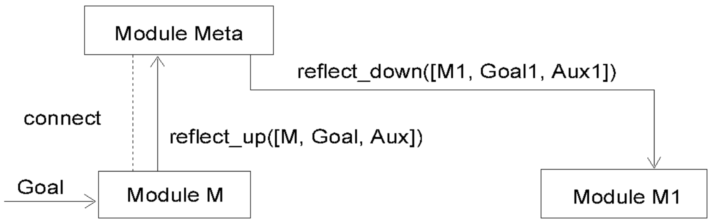

Figure 1: Proof of a goal in the module M.

Strictly speaking, we have discussed only two level logic programs so far. However, it is apparent that the concept of introducing a meta-level to a given program level can be applied to the meta-level as well, yielding a meta-meta-level, etc. While conceptually this appealing idea is quite simple, there are certain more practical issues which require careful consideration.

We wish to underline that what we are facing at this stage is in some sense a design problem. It requires design decisions, based on considerations of various options. The problem may not have a unique solution.

A multilevel logic (Prolog) program is a modular logic (Prolog) program in which modules can be mutually interconnected by defining the relation *connect*. The relation *connect* is used to establish program levels. At the lowest (i.e., object, or program) level, program modules are defined. At higher levels, modules are defined which determine the way goal is proved in program modules. Both program and meta-level modules are represented in the same way, and therefore further meta-levels are naturally possible.
For a detailed description of the axiomatic system, see Appendix.

3 Combining Modules in Multilevel Logic Programming

One of the possible advantages of the multilevel logic programming is that it offers means for structuring of programs. Mere dividing of a program into modules at several levels introduces a rudimentary structure to the program. There are various ways of combining modules in a logic program structured to multiple levels.

3.1 Union of Modules

Combination of modules is a union of clauses of modules being combined.[10,4] Formal semantics of the operator '*' which combines two modules is defined by Brogi[5]. It is commutative. We shall use also the operator $\rightarrow$. Expression $E \rightarrow G$ denotes a relation between a composition of modules E and a goal G inferred from them. Throughout the text we will assume the following operators definition in the respective modules:

```
  :- op(950, yfx, '*').        %union operator
  :- op(950, xfy, '->').       %inference operator
```

In order to implement the union of modules using the multilevel Prolog programming technique, we define the module *meta_union*:

```
 meta_union ismod
  {   reflect_up([M, (P1 * P2) -> Goal, Aux]) :-       %clause1
            !, reflect_up([M,P1 -> Goal,[P2 | Aux]]).
      reflect_up([M, P -> Goal, Aux]) :-               %clause2
            !, reflect_up([M, Goal, [P | Aux]]).
      reflect_up([_, Goal, Composition]) :-            %clause3
            member(Module, Composition),
            reflect_down([Module, Goal, Composition])   }.
```

The term AUX is used to store a set of modules which form a composition of modules active in inferring the goal G. The order in which the modules are tried is determined by the predicate `member/2`.

3.2 Closed Modules

This combination of modules allows to infer only those formulae which can be inferred from single modules.[18,9,4] Predicate definitions in modules are considered local. Modules are *closed* (as opposed to *open* modules in the union).

In order to implement the local strategy using multilevel Prolog programming technique, we define the module *meta_closed*:

```
meta_closed ismod
 {   reflect_up([M, (P1 * P2) -> Goal, Aux]) :-
           !, reflect_up([M, P1 -> Goal,[P2 | Aux]]).
     reflect_up([M, P -> Goal, Aux]) :-
           !, reflect_up([M, Goal, [P | Aux]]).
     reflect_up([_, Goal, Composition]) :-
           member(Module, Composition),
           reflect_down([Module, Goal, [Module]])          }.
```

The term AUX is used to store the current (closed) module (its clauses are considered in order to prove the *Goal*). Downward reflection is attempted in the $Module$ with AUX set to the one element list $[Module]$, i.e. all subgoals are to be proved only with clauses defined in the $Module$. When an attempt to prove *Goal* fails, next module from the $Composition$ is chosen and a downward reflection is attempted again up to *Goal* is proved or all the modules from the list $Composition$ were used.

3.3 Modules with Imported Predicates

Let us consider the operator *close* with two arguments. The first one is a module. The second one is a set of predicates imported by that module. A formula $close(M, Imp)$ divides predicates in the module M into two parts:

- imported predicates, i.e. those included in the set Imp. They stay open with respect to actual composition of modules,
- not imported predicates, i.e. those not included in the set Imp. They are closed and can be applied only within the module M.

The definition of the operator *close* is in fact a generalization of both the open and closed modules. A completely closed module is expressed by the formula $close(M, [])$, i.e. the set of imported modules is empty. On the other hand, an open module is expressed by the formula $close(M, preds(M))$, where $preds(M)$ denotes a set of all functors of predicates in M.

When defining the described strategy of combining modules, it is necessary to maintain both the global and local compositions. The local composition is used when applying not imported predicates in a closed module. The global composition is used when imported predicates are applied. The global and local compositions are represented by the term AUX.

In order to implement the described strategy using multilevel Prolog programming technique, we define the module *meta_composition*:

```
meta_composition ismod
 { reflect_up([M, (P1 * P2) -> Goal, Aux]) :-
      !, reflect_up([M, P1 -> Goal, [P2 | Aux]]).
   reflect_up([M, P -> Goal, Aux]) :-
      !, reflect_up([M, Goal, [[P | Aux], [P | Aux]] ]).
   reflect_up([_, Goal, [Global, Local] ]) :-
      Goal =.. [Name|_],
      member(close(Module, Import), Local),
      ( member(Name, Import),            %Goal in import list
        ( Global \= Local,
          reflect_up([Module, Goal, [Global, Global]])  ;
          Global = Local,
          reflect_down([Module, Goal, [Global, Global]])  )
        ;
        not( member(Name, Import)),
        reflect_down([Module, Goal,
                       [Global,[close(Module,Import)]] ]) ) }.
```

3.4 Contextual Programming

Other proposals for structuring logic programs adopt more complex policies than the open and the closed modules composition. Typical example is a definition of nested modules, or a notion of blocks.[9] They are inspired by conventional programming languages. Moreover, we can mention different ways of inheritance.[17] In all of these cases, compositions of modules introduce an ordering among programs.

Statically, a program is a set of modules. Dynamically, goals are solved in changing sets of modules (contexts). Contextual programming has been formally described by Monteiro and Porto.[20] We shall concentrate on representing knowledge about contextual programming. Our goal is to show suitability of the multilevel Prolog for implementing different contextual programming strategies.

A context is an ordered set of modules which can change during the process of proving a formula. Contrary to standard logic programming, where definitions of predicates are given statically and cannot be changed, in contextual programming definitions of predicates are no longer static and depend on an actual context used in a proof.
There are various approaches that differ in determining the way both

- the definition of a predicate which is applied during a proof of the formula in the given context. In fact, the previously presented strategies can be considered as special cases of contextual programming, and
- the procedure to follow after applying a particular clause. Two options are to be mentioned: (i) the inference continues in the actual context (evolving policy), (ii) the inference continues in a modified context that is restricted to modules that appear in the original context after the module used in this inference step (conservative policy).

In order to demonstrate the representation of contextual programming strategies using the multilevel Prolog programming technique, we define the module *meta_context_conservative_policy*:

```
meta_context_conservative_policy ismod
 {   reflect_up([M, Context -> Goal, Old_context]) :-
            !, reflect_up([M, Goal, Context]).
     reflect_up([_, Goal, Context]) :-
            member_after(Module, Context, After),
            reflect_down([Module, Goal, After]).

     member_after(E,[E|R],[E|R]).
     member_after(E,[_|R],A):-  member_after(E,R,A)  }.
```

In the module *meta_context_conservative_policy*, there is used only the operator `'->'`. It serves for context switching. In case of one element list, using the strategy equals a proof of a goal with considering clauses defined in a different module (i.e., call).

3.5 Inheritance

Simple inheritance can be expressed as a special case of the contextual programming. A context is just an explicit representation of one path of an inheritance hierarchy tree. The first element in the context represents a leaf, the last element is a root. If the context is $[M_n, \ldots, M_i, \ldots, M_1]$, the modules before M_i are called sub-modules and the modules after M_i are called super-modules to the module M_i.

The inheritance strategies can be grouped into following classes:[14]

- a syntactic inheritance: to prove a given goal, a super-module is attempted if there is not any predicate with the same functor and arity as the goal in the actual module,
- a unification: to prove a given goal, a super-module is attempted if there is not any predicate unifiable with the goal in the actual module,
- a success/failure: to prove a given goal, a super-module is attempted if its proof in the actual module has failed.

We present meta-modules that implement the described strategies. It is assumed that the inheritance relations between modules are expressed explicitly as a fact in the form `super(M2, M1)`. The inheritance tree is defined by the predicate `ancestor/2`. In the meta-modules we use the fact that each module in an ancestor tree is super-module itself.

```
meta_syntactic_inheritance ismod
 {   reflect_up([Module, Goal, AUX]) :-
          transform(Goal, Goal1),
          ancestor(Module, Ancestor),
          reflect_down([Ancestor, clause(Goal1, _), AUX]),
          !, reflect_down([Ancestor, Goal, AUX])             }.

meta_unification_inheritance ismod
 {   reflect_up([Module, Goal, AUX]) :-
          ancestor(Module, Ancestor),
          reflect_down([Ancestor, clause(Goal, _), AUX]),
          !, reflect_down([Ancestor, Goal, AUX])             }.

meta_success/failure_inheritance ismod
 {   reflect_up([Module, Goal, AUX]) :-
          ancestor(Module, Ancestor),
          reflect_down([Ancestor, Goal, AUX])                }.
```

The predicate `transform(Goal, Goal1)` generates the term $Goal1$, which has the same functor as $Goal$ and all the arguments of which are unbound variables.

4 Structuring the Meta-Level Knowledge

Developing the meta-knowledge (i.e., knowledge on program modules, for instance a definition of how to combine them) is a process with similar characteristics as developing program modules. When developing large systems with complicated strategies of combining program modules, we face similar problems at meta-levels as at the object level. Structuring the meta-knowledge is thus another step in supporting the program development.

Having the above presented multilevel Prolog programming in mind, we propose the following ways of the structuring meta-knowledge: (i) distributing the meta-knowledge into separate modules, with each module defining one particular strategy, (ii) dividing the meta-knowledge that defines one strategy into several modules at one level, and defining a way of combining them.

An example of the former way is a combination of the context switching and the simple inheritance method. At the first meta-level, there is defined the module *meta_context*, at the second meta-level there is defined a module *meta_meta_inheritance*. Desired behaviour will result from proper defining of the *connect* relation, see Figure 2.

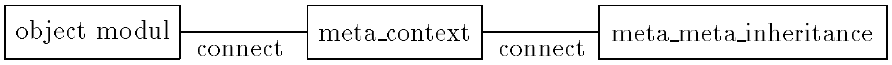

Figure 2: Relation *connect* for structuring the meta-knowledge.

The module *meta_context* is implemented at the first meta-level. It includes only knowledge related to the context switching method. The module *meta_meta_inheritance* defines not only an inheritance mechanism, but also a behaviour during proving any goal within the module *meta_context*.
The desired behaviour is described by the following knowledge piece:

if *there is an attempt to reflect down*
  then *apply the inheritance strategy*
  else {*no additional knowledge is needed to the method defined in the meta-module from which the process switched to the actual level*}

The other proposed way of structuring the meta-knowledge makes it possible to decompose more complicated strategies and to represent them in several meta-modules. Moreover, it is possible to define a way of combining such modules. The approach is suitable also in the case more modules contain equal parts.

An example of this approach is a representation of the knowledge on the closed and open modules. We have two modules *meta_closed* and *meta_union* which contain common parts (the first two clauses of the predicate *reflect_up*).

Instead of defining modules as the two ones above, the same behaviour can be defined by connecting program modules (the definition of the relation *connect*) either with a composition [*meta_common*, *meta_closed*1], or with a composition [*meta_common*, *meta_union*1] depending on which strategy is considered. The module *meta_common* contains clauses shared by *meta_closed* and *meta_union*.

This approach requires defining a way of interpreting the relation *connect*. A possible approach is to define a way of combining the meta-modules (i.e. meta-meta-knowledge) in a special module in the form of predicates *connect_up* and *connect_down* with an interpretation similar to those of *reflect_up* and *reflect_down*. Interpretation of the relation *connect* is thus separated from a description of a solution of the given problem.

In this case, it is necessary to include a knowledge piece: *"global strategy is defined by a sequence of clauses as defined in the modules meta_common and meta_union1 (in the indicated order)"* in the definition of the relation *connect* among program modules and meta-modules. It is not directly related to the program nor to the problem being solved. It is a meta-knowledge related only to the open modules composition strategy. It is therefore more natural to express it in a meta-meta-module. The multilevel Prolog allows us to define it in the following way:

```
meta_meta ismod
 {  reflect_up([_, reflect_down([Module, Goal, Aux]), _]) :-
           !, reflect_down([Module, Goal, Aux]).
    reflect_up([_, Goal, Aux]) :-
           member(M, [meta_common, meta_union1]),
           reflect_down([M, Goal, Aux])               }.
```

The relation *connect* is defined according to the Figure 3.

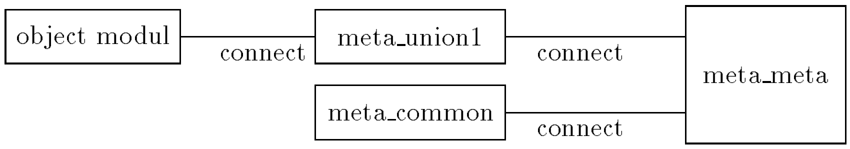

Figure 3: Relation *connect* for structuring meta-knowledge.

5 Conclusions

We have presented our proposal of a multilevel logic programming technique. Our approach is based on the reflection technique.[13] In our approach, the technique is extended by the possibility of forming and processing meta-levels to an arbitrary height. We have also devised several patterns of switching the computation among meta-levels. The proposal has been implemented in Prolog. The prototype is meta-interpreted.

The proposal of a multilevel logic programming technique is to be considered a contribution to methods of structuring logic programs. Organizing programs into (meta-)levels refers to knowledge content. A need to structure programming knowledge according not only to abstraction and generality levels, but to meta-levels as well has been stressed by Návrat.[21] The fact that also meta-knowledge can be written in modules aids to modifiability and reuse. Meta-level can be used to write a module defining various ways of processing goals from the object level. Meta-meta-level would be suitable to write a module defining method of selecting the proper way of processing.

Our approach allows also to connect an object level program to several modules at the same meta-level. Similar question was tackled by Sterling[26] who proposed two strategies of combining meta-interpreters.

Multilevel Prolog programming can be used with advantage whenever a problem domain knowledge is available. After careful analysis, several layers of knowledge can usually be recognized. There is knowledge of the problem itself. There is also another kind of knowledge which describes structure and properties of objects, relations, including ways of solving problems. This is meta-knowledge. It can and, in fact, it should be structured just as any other knowledge should. Knowledge is better captured, understood, manipulated, and applied if structured into interconnected units. But supposing e.g., there are several problem solving methods defined at the meta-level, it is very likely that there is also another kind of knowledge available, viz. the one evaluating their respective suitability, applicability, etc. This is already a meta-meta-level knowledge. It can be extremely useful in deciding which method to apply to a particular problem instance. It is quite clear that structuring knowledge according to such content, i.e. semantically related hierarchies can be potentially at least as fruitful as those more syntactically oriented approaches.

We applied our programming technique to a problem of software version selection. It is a problem from the area of software configuration management.[22] Our concern for software version selection is part of a larger project aimed at developing a method for building a software configuration.[3] Another application is in representation of a rule-based query optimizer for object-oriented databases.[2] Multilevel logic programming is used to model both query rewriting and planning, as well as search strategies.

Appendix

Let P be a multilevel logic program, G be a conjunctive formula, A, A' be atomic formulae, τ, δ, σ be substitutions, ϵ be an empty substitution. Composition of two substitutions is denoted by concatenation. $G\tau$ denotes an application of the substitution τ to G. Let $mgu(A, A')$ denotes the most general unifier of two atomic formulae A and A'. Let $mod(P)$ denotes set of names of modules of program P and $|M|$ denotes set of clauses defined in module M.

A goal G is provable in a multilevel logic program P in a module named M with substitution τ if there exists a proof of the formula $P \vdash \tau(M, G, [\,])$.

Proof of a formula G in a module M of a multilevel logic program P can be written as a sequence of formulae $P \vdash \tau_i(M_i, G_i, AUX_i)$, where M_i is name of a module in program P, G_i is a goal, AUX_i is a term, and τ_i is a substitution. Initially, we start from an empty auxiliary memory AUX, i.e. $AUX_1 = [\ ]$. Next formula of a proof is obtained by applying a suitable inference rule. Goal is proved if a formula is inferred with $true$ in place of a goal after a finite number of steps. The inference rules are written in the form

$$\frac{premises}{conclusion}$$

Inference rules:

1. True I

$$\frac{}{P \vdash \epsilon(M, true, AUX)}$$

2. True II

$$\frac{}{P \vdash \epsilon(M, true)}$$

3. Conjunction I

$$\frac{P \vdash \tau(M, A, AUX), P \vdash \delta(M, G\tau, AUX)}{P \vdash \tau\delta(M, (A, G), AUX)}$$

4. Conjunction II

$$\frac{P \vdash \tau(M, A), P \vdash \delta(M, G\tau)}{P \vdash \tau\delta(M, (A, G))}$$

5. Atomic formula I (no reflection)

$$\frac{\begin{array}{c} M \in mod(P), A' :- G \in |M|, \\ \tau = mgu(A, A'), \\ P \vdash \delta(M, G\tau) \end{array}}{P \vdash \tau\delta(M, A)}$$

6. Atomic formula II (upward reflection)

$$\frac{\begin{array}{c} Meta \in mod(P), M \in mod(P), \\ P \vdash \sigma(M, connect(Meta)), \\ A' :- G \in |Meta|, \\ \tau \in mgu(reflect\_up([M, A, AUX]), A'), \\ P \vdash \delta(Meta\sigma, G\tau, AUX1) \end{array}}{P \vdash \tau\delta\sigma(M, A, AUX)}$$

7. Atomic formula III (connect not defined)

$$\frac{\begin{array}{c} M \in mod(P), \\ \neg(Meta \in mod(P) \wedge P \vdash \sigma(M, connect(Meta))), \\ A' : -G \in |M|, \\ \tau = mgu(A, A'), \\ P \vdash \delta(M, G\tau, AUX) \end{array}}{P \vdash \tau\delta(M, A, AUX)}$$

8. Atomic formula IV (downward reflection)

$$\frac{\begin{array}{c} Meta \in mod(P), \\ \neg(Meta1 \in mod(P) \wedge P \vdash \sigma(Meta, connect(Meta1))), \\ M \in mod(P), A' : -G \in |M|, \\ \tau = mgu(A, A'), \\ P \vdash \delta(M, G\tau, AUX) \end{array}}{P \vdash \tau\delta(Meta, reflect\_down([M, A, AUX]), AUX1)}$$

The rules 2, 4, 5 define procedural semantics of a modular logic program. These rules are necessary in order to determine which module is a meta-module with respect to a given program module (the relation *connect*).

Proof at a meta-level has the same procedural semantics as at the program level. If a module M is *connect*ed to some other module, upward reflection occurs to next higher level (cf. rule 6). In particular, upward reflection can occur during an attempt to satisfy a goal *reflect_down*, too. However, if there is not *connect*ed any module to a given module during an attempt to satisfy a goal *reflect_down*, reflection occurs towards a level determined by a parameter of the term *reflect_down* (cf. rule 8).

References

1. Ch. Beierle in *Proc. Sorts and Types in AI,* ed. K.H. Bläsius, V. Hedstück and C.R. Rollinger (Springer-Verlag, 1990).
2. M. Bieliková *et al, Revue ing. des systèmes d'information*, to appear.
3. M. Bieliková and P. Návrat, *Knowledge Based Systems*, **9** (1), 61–65 (1996).
4. A. Brogi, E. Lamma and P. Mello, *J. of Logic Computation*, **3** (9), 1–25 (1992).
5. A. Brogi *et al, ACM Trans. on Program. Lang. and Systems*, **16** (4), 1361–1398 (1994).

6. K.L Clark and F.G McCabe, *Micro-Prolog: Programming in Logic* (Prentice-Hall Int., 1984).
7. D. Friedman and M. Wand in *ACM Symp. on LISP and Func. Prog.* (1984).
8. H. Gallaire in *AAAI Conf.* (1986).
9. L. Giordano, A. Martelli and G.F. Rossi, *Software - Concepts and Tools*, **15** (3), 125–145 (1994).
10. L. Giordano, A. Martelli and G.F. Rossi, *Theoretical Computer Science* (1992).
11. R.A Kowalski, *Logic for problem solving* (North-Holland, 1979).
12. K. Kwon, G. Nadathur and D.S. Wilson in *Extension of Logic Programming ELP'92*, ed. E. Lamma and P. Mello (Springer-Verlag, 1993).
13. E. Lamma, P. Mello and A. Natali, *Software - Practice and Experience*, **21** (6), 603–624 (1991).
14. L. Leonardi, P. Mello and A. Natali, *J. of Object-Oriented Programming*, **2** (3), 20–28 (1989).
15. F.G McCabe, *Logic and Objects* (Prentice Hall, 1992).
16. M. Meier in *Extension of Logic Programming ELP'92*, ed. E. Lamma and P. Mello (Springer-Verlag, 1993).
17. P. Mello in *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, ed. M. Simi, M. Lenzerini and D. Nardi (John Wiley & Sons Ltd, 1991).
18. P. Mello and A. Natali in *Proc. ESOP 86*, ed. B. Robinet and R. Wilhelm (Springer-Verlag, 1986).
19. D.A. Miller, *J. of Logic Programming*, **6**, 79–108 (1989).
20. L. Monteiro and A. Porto in *6th Int. Conf. on Logic Programming*, ed. G. Levi and M. Martelli (The MIT Press, 1989).
21. P. Návrat, *Information and Software Technology*, **38** (1), 37–46 (1996).
22. P. Návrat and M. Bieliková, *Software - Concepts and Tools*, **17**, 40–48 (1996).
23. G. Razek, *ACM SIGPLAN Notices*, **27** (12), 66–70 (1992).
24. D.T Sannella and L.A Wallen, *J. of Logic Programming*, (12), 147–177 (1992).
25. U. Schreiweis *et al, ACM SIGPLAN Notices*, **28** (4), 21–28 (1993).
26. L. Sterling and R.D. Beer, *J. of Logic Programming*, 163–178 (1989).
27. D. Xu and G. Zheng, *ACM SIGPLAN Notices*, **30** (1), 5–10 (1995).
28. C. Zaniolo in *Int. Symposium on Logic Programming* (1984).