A Multi-Level Logic Programming Model of a Query Optimizer

Mária Bieliková
Slovak University of Technology,
Ilkovičova 3, 812 19 Bratislava, Slovakia
E:mail: bielik@elf.stuba.sk
WWW: http://www.dcs.elf.stuba.sk/~bielik

Béatrice Finance
CNRS-PriSM Laboratory, Univ. of Versailles-St-Quentin,
78035 Versailles, France,
E:mail: Beatrice.Finance@prism.uvsq.fr
WWW: http://www.prism.uvsq.fr

Pavol Návrat
Slovak University of Technology,
Ilkovičova 3, 812 19 Bratislava, Slovakia
E:mail: navrat@elf.stuba.sk
http://www.elf.stuba.sk/~navrat

Abstract

The paper describes a rule-based query optimizer for object-oriented databases. The originality of the approach is through a multi-level logic programming used to model the variety of knowledge contained in the query optimizer in an explicit, declarative and transparent way. Our approach offers means of abstraction for expressing various kinds of knowledge involved in a query optimizer. It also offers techniques for structuring them according to both generality levels and knowledge content, i.e. meta-levels. We present a programming technique that allows to write modules which can be at various meta-levels. To illustrate these ideas, we show how multi-level programming can be used to model a query optimizer for an object-oriented database. Among the various kinds of knowledge involved, we have (besides the queries themselves - first or object level) techniques for query manipulations and transformation, as well as cost models (second or meta- level), techniques for combining transformations, search strategies, techniques for cost model selection (third or meta-meta- level), and optimization plans (fourth level). The optimizer architecture based on this model is presented.

1 Introduction

Query optimization in extended relational, object-oriented and deductive systems is one of the key issues in the current database literature. Traditionally, query optimization translates a high-level user query into an efficient plan for accessing the database facts. Its essence consists in finding an execution plan that satisfies a given criterion (e.g. minimizes a cost function). Query optimization can be divided in two phases [13]: *query rewriting* transforms queries into equivalent simpler ones with better expected performance and *query planning* primarily determines the method for accessing objects. Several search strategies have been proposed both for query rewriting and planing. For query rewriting, a strategy based on heuristics is usually chosen. Rules are ordered and alternative plans are not memorized. In the opposite, query planning memorizes many alternatives to find the plan that has the minimum cost. The diversity of the tasks that should be integrated in a query optimizer makes it one of the most complex components to write in a DBMS.

In the past, query optimizers mainly relational ones [25, 1] followed a 'black-box' approach. The optimizer knowledge was procedural making it difficult to evolve. Recently, extensible optimizers have been proposed [12, 13, 22]. The key idea was to generate a query optimizer from rules for transforming plans into alternative

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

plans. Many rule languages have been proposed: term rewriting, functional or algebra equivalences. To control the rule evaluation, many search strategies have been also proposed including variations of enumerative search [23] and randomized search [14]. The rule-based approach is very promising; however it is hard to apply, because it is very difficult to organise and to combine the knowledge of the query optimizer.

In this paper, we propose to follow a multi-level logic programming to model the query optimizer. Methods and techniques for query optimization are based on various kinds of knowledge. Among them, we find: (1) techniques for query manipulation and transformation, (2) control strategies, (3) cost models, (4) description of classes of queries, (5) optimization goals, and (6) fine and large granularity of optimization. They interact in various ways; moreover, not all of them are at the same level. For example, a control strategy refers to elements of the set of transformation rules. Transformation rules in turn refer to descriptions of classes of queries.

Our focus is on a uniform formalism used to represent different approaches. To illustrate our ideas, we propose a possible architecture for the object-oriented query optimizer that was developed as a part of a COPERNICUS project [3]. This query optimizer has been specified by the PRiSM Laboratory [6]. Optimization rules are described by using OFL (Object Functional Language). OFL is a target object language for OQL-like query compilers introduced in [9]. OFL programs describe the traversals of collections in the execution world. In the traversal, functional predicate and projection expressions can be applied. By using the OFL rewriting formalism the search space explored by the classic *query rewriting* and *query planning* phases of an object optimizer are uniformly described by OFL programs.

The goal of our work is not to introduce yet another technique for the optimization of object-oriented queries, but rather to devise a representation framework for various knowledge involved in techniques for rewriting of object- oriented queries. Our approach aims to achieve the following properties regarding the optimization:

- recognition of hierarchical levels within the relevant knowledge,
- structuralization of optimization knowledge which enables better understanding, readability and maintenance,
- extensibility of the optimizer (possibility to improve optimizer as the field evolves),
- easy combining of different methods and techniques by specifying alternative modules and meta-modules which control their use in the optimization process, and
- development and experimental testing of new approaches (e.g. new transformation rules and strategies which are domain dependent).

2 A Programming Technique for Implementing Multi-level Logic Programs

One of the crucial problems of logic programming is the lack of concepts, and consequently of language constructs, for structuring, modularity, sharing and hiding, etc. which are all important means to manage the complexity of a particular domain. The desire to have in Prolog means for dividing a program into smaller relatively separated and independent units with transparent minimal interfaces has been responded by several authors [21, 11, 18, 16, 15, 19, 26].

Problems are encountered when trying to combine logic databases (modules). Several approaches have been tried e.g., inheritance [19], context switching [16], introducing implication into goals [11], different definitions of visibility of atoms [18], using abstraction in separating the logic database from the concrete implementation by specifying required resources and produced results [21]. The 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 any 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 the given problem.

The above described difficulty is often approached with the meta-programming technique. Essentially, meta-programming is a technique that allows to treat programs as data. When meta- programming, one is able to define various modifications of the way a program is processed. It is suitable for implementing different ways of communications of logic databases [5].

A usual straightforward 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 [24].

An alternative approach is to allow direct access to some specific parts of the status of the abstract program processing machine. 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 parts which have to be modified. The approach is not entirely new, cf. [8], but not so much attention has been paid to it as 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.

Rather than (meta-) interpreting the overall behaviour of the abstract machine, some parts of the machine's state are made available to be accessed and manipulated directly through 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) [16].

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, G, AUX)*, where M is a current module, G is a 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. In fact, this allows us to apply the reflection concept to a program designed into many levels, as we shall show later. 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 module M. When the module *Meta* becomes the 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, G, 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, G1, AUX1])*. This causes an object level computation to start in the module *M1* aiming to prove the goal *G1*. The attempt can again 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 a meta-level. If it fails, the failure is reported to *reflect_up* goal at a meta-level which fails, too. Described computation is illustrated in Figure 1.

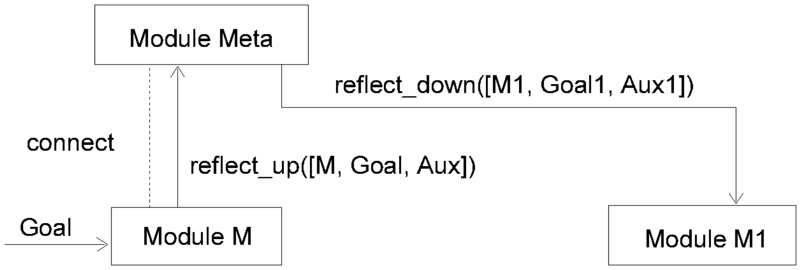

Figure 1: Proof of a goal in the module M.

We have deliberately not explained the role of the third part AUX of the visible state. Syntactically, it is a term to be processed at a meta-level. Semantically, the choice is left open to be determined in accordance with the application domain of problems being solved.

Strictly speaking, we have discussed only two level logic programs so far. However, it is apparent that the concept of introducing meta-level to a given program level can be applied to the meta-level as well, yielding meta-meta-level, etc. While conceptually this appealing idea is quite clear, there are certain more practical issues which require careful consideration [2].

A multi-level 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 the program levels. At the lowest (object, or program) level, program modules are defined. At higher levels, modules are also defined; they determine the way a goal is proven by the program modules. Both program and meta-level modules are represented in the same way, and therefore further meta-levels are naturally possible.

3 Query Optimizer Architecture

In this section, we try to identify the main components (modules) of a query optimizer together with examples of optimizer representation by a multi-level logic program.

In describing the particular levels of the optimizer's architecture, we start with a query which is to be considered the lowest level, i.e. *level 1.* Because a query written in the functional language OFL is syntactically a term in a logic programming language, a query can be written in Prolog directly as a fact. Reflecting the requirement to allow independent manipulation with parts of a query, it seems reasonable to represent them by facts as well. The level of decomposition of the query has been derived from the basic functions of the OFL.

At the *level 2*, there are transformation rules which define operations of the optimizer.Various sets of transformation rules which perform complex manipulations of queries can be defined. In fact, we could have just one set of transformation rules, with each particular technique defining its relevant subset. For example:

- a strategy of reducing the cost of a query execution would employ a cost-guided technique. It uses syntactical transformation rules directed by cost estimations of the operations and data involved in the query;
- another strategy of reducing the cost of a query execution would employ a semantic query optimization technique. It uses semantically equivalent transformations, such as eliminating unnecessary joins, and adding/deleting redundant beneficial/nonbeneficial selection operations;
- another strategy of reducing the cost of a query form would employ either a join conversion or a join reordering technique. The technique of join conversion uses rules that transform an arbitrary query into a canonical join form. The technique of join ordering manipulates queries that are expressed in terms of join operations to determine good join orderings and access methods.

A rule is represented as a Prolog term with at least two arguments, which specify the rewriting ($\tau_1 \rightarrow \tau_2$). The third argument is defined for additional characteristics of representation such as constraints, methods, differential cost computations.

The process of query optimization can be made more efficient by introducing an appropriate structure within the set of transformation rules. Particular subqueries can frequently be transformed by different subsets of transformation rules. Rule partitioning into subsets along with an explicit representation of subqueries, facilitates working with components of a query. There can be represented at the same hierarchical level different cost models which define methods of query evaluation.

The next level of the architecture description is the *level 3*; it comprises search strategies. In the literature several strategies are proposed including variations of enumerative search [23] and randomized search [14, 7, 10]. Our approach offers means for representing the whole class of well-known strategies as a special case of a data driven strategy for state space search as formulated in artificial intelligence.

A search strategy can be very briefly described in general as follows:

Search from the state S_i

1. find successors to the state S_i in the state space (the set N)
2. reduce the set of successors N according to the given criterion, resulting in a set NR
3. test if the goal state is in the set NR
   – if it is, stop
   – if it is not, continue with the step 4

4. select one element S_{i+1} from the set NR and search from the state S_{i+1}.

The search starts with the state S_i as the initial state. Particular strategies such as iterative improvement or simulated annealing can be derived from the above strategy by specifying the way of determining the successors, the way of reducing the set of successors, or the way of selecting the succcessor to be processed next. Particular strategies are represented by modules which are coupled with the module that represents the generic strategy. Those strategies which are based on different principles, such as the so called transformation free strategy [10] are represented by a special module.

For illustration we show a representation of the generic strategy from which e.g., simulated annealing can be derived.

```
search_strategy ismod            %generic strategy
{  strategy([Stop_Cond|_], Searched_Space, Min_State) :-
         call(Stop_Cond),!,            %stop condition satisfied
         get_minimal_state(Searched_Space, Min_State).

   strategy(Parameters, Searched_Space, Min_State) :-
         get_current_state(State, Searched_Space),
         do_one_step(Parameters, New_Parameters, State, New_State),
   strategy(New_Parameters, [New_State|Searched_Space], Min_State)    }.

simulated_annealing ismod       %specific strategy
{  reflect_up([_,do_one_step([Temp|Rest_Par],[New_Temp|Rest_Par],State,New_State),AUX]):-
         do_one_step_aux([Temp|Rest_Par], State, New_State),
         reduce(Temperature, New_Temp).

   do_one_step_aux([_, Equilibrium_Cond|_], Chosen_State, Chosen_State) :-
         call(Equilibrium_Cond), !.
   do_one_step_aux([Temperature|Par], Current_State, Chosen_State):-
         generate_neighbors(Current_State, List_of_neighbors),
         select_neighbor(List_of_neighbors, Neighbor),
         (  is_better([Temperature|Par], Neighbor, Current_State),
            !, do_one_step_aux(Temperature, Neighbor, Chosen_State)
            ;
            do_one_step_aux(Temperature, Current_State, Chosen_State ))  }.
```

When goal *strategy(Optimiz_Par, Searched_Space, Best_Space)* is to be solved in the module *search_strategy* the specific strategy is evaluated by a connection between generic module and specific module. Proof of the goal *do_one_step/4* is reflected to the level represented by the module *simulated_annealing* if such connection is defined. How to perform one step in state search is defined in this module. Let us note that the predicate *select_neighbors/2* in simulated annealing algorithm can be satisfied repeatedly. When backtracking, it returns particular neighbors generated and represented in a list.

Now let us turn our attention to the way of implementing a comparison between the different states. For experimental purposes, we can have several cost models which may depend on a database machine, or on a particular database. Sometimes, appropriate algebraic measures can be used e.g., a number of predicates in the execution plan, an estimated number of collection being visited, etc. Moreover, there are alternative criteria for acceptance of a state. In the transformation-free algorithm, the state with a lower cost is simply accepted. On the other hand, the simulated-annealing algorithm can accept a state with a higher cost according to a defined probability which decreases as the time progresses.

The definition of the predicate *is_better/3* should be separated, mainly due to the possibility of different interpretation of the relation *is_better*. An attempt to satisfy the goal *is_better(Optimiz_Par, State1, State2)* always causes reflection to a meta-level represented by the module *meta_comparing*. Here a decision is made about the way the relation is interpreted. The decision is determined by the control strategy being currently evaluated. Moreover, the first parameter can identify a measure for cost determination. All this requires, however, that a connection has been established between level represented by modules for search strategies, e.g. *simulated_annealing* and the above mentioned meta-level.

```
meta_comparing ismod
{ reflect_up([iterative_improvement,is_better(Par,S1,S2),AUX]):-
     cost(Par,S1,Cost_S1), cost(Par,S2,Cost_S2),
     Cost_S1 < Cost_S2.

  reflect_up([simulated_annealing,is_better([Temp|Rest_Par],S1,S2),AUX]):-
     cost(Rest_Par, S1, Cost_S1), cost(Rest_Par, S2, Cost_S2),
     ( Cost_S1 < Cost_S2
       ;
       Cost_S1 >= Cost_S2,
       Probability is e^((Cost_S1 - Cost_S2)/Temp),
       appearance(Probability)  )
             ......

   %similarly for other control strategies        }.
```

This approach has an advantage which consists in separating the interpretation of the *is_better* relation from its application. It can easily be accessed, modified, or enhanced. The similar way is used for representing other concepts in optimization, including transformation rule representation (each set of rules in one module), cost model definition, method for random state generation, etc.

Along with strategies at the same level of abstraction, knowledge pieces on how to combine modules of transformation rules are also represented. There are several well-known strategies of combining program modules, as studied by the programming and software engineering community [19]. At the same level, various ways of evaluating queries based on different cost models can be represented as well. The cost model can be altered during execution.

Ways of applying strategies of state space search are defined at the level 4. At this level, strategies are treated as objects to be referred to. Here, there can be represented e.g., optimization plans i.e., how and when to apply a particular strategy. This module can use for example a control strategy similar to the ones used in space searching by artificial intelligence techniques such as hill climbing. Here a decision can be made about an application of different control strategies used in query optimization e.g. try iterative improvement with stopping condition which constrains the number of local minima found and after that try the transformation free technique. Finally select the best solution.

The optimizer architecture with its modules outlined above can be depicted as in Figure 2.

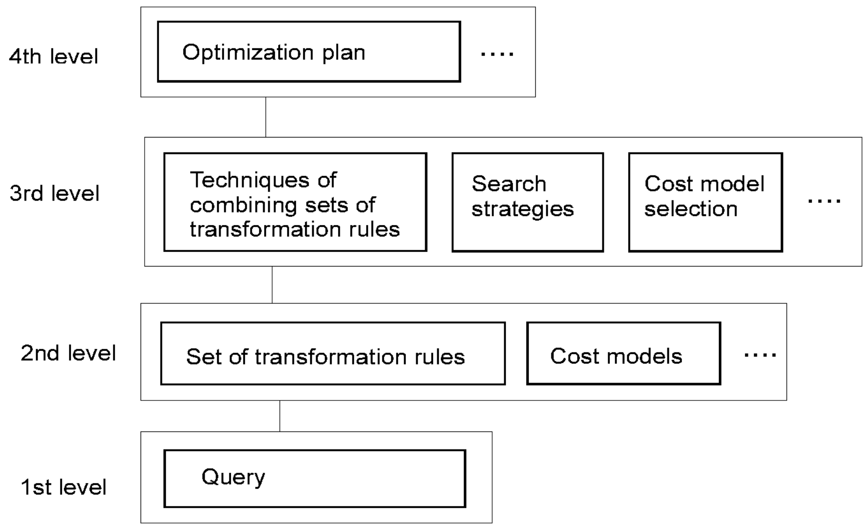

Figure 2: Optimizer architecture.

4 Combining Modules in Multi-level Logic Programming

Above, we have described several examples of possible modules included in the optimizer. Relationships between them were established by the relation *connect*. There exist different possibilities for combining these modules which are similar to those used in conventional programming. They can be used in an optimizer and easily represented by the proposed multi-level Prolog. There are several well-known methods or strategies for combining modules, such as global strategy, local strategy, context switching, inheritance. A detailed description of these strategies together with demonstrating on how these and possibly other methods can be implemented within our multi-level logic programming framework is reported in [2]. Now we briefly describe the main ideas.

Global strategy

The combination of modules is the clause union of the modules being combined. The formal semantics of the operator '*' which combines two modules and which is commutative are defined in [5].

Local strategy

The combination of modules allows to infer only those formulae which can be inferred from single modules [11]. Predicate definitions in modules are considered local. Modules are 'closed' [4]. The use of the local strategy to combine modules is quite restricted, due to the fact that predicates cannot be imported.

Modules with imported predicates

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

- 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 global and local strategies. 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 the set of all functors of predicates in M. The declarations of imported predicates are dynamic. Modules can be used in different ways, just by modifying the second argument in *close*.

Contextual programming

Other proposals for structuring logic programs adopt more complex policies than the global and the local ones. Typical examples are the definition of nested modules, notions of blocks [11]. They are inspired from conventional programming languages. Moreover, we can mention different ways of inheritance [19]. In all these cases, the compositions of modules introduce an ordering among programs, often by stacks.

Statically, a program is a set of modules. Dynamically, goals are solved in the changing sets of modules (contexts). Contextual programming is formally described in [20]. A context is an ordered set of modules which can change during the process of proving a formula. Contrary to standard logic programming, where predicate definitions are given statically and cannot be changed, predicates definitions are in contextual programming no more static and depend on the actual context used in the proof. There exist various ways of determining the actual definition of a predicate with respect to the actual possible context. In fact, the previously presented strategies can be considered as special cases of contextual programming.

Inheritance

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

The inheritance strategies can be grouped into the following classes [17]:

- *syntactic:* 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,
- *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,
- *success/failure:* to prove a given goal, a super-module is attempted if its proof in the actual module has failed.

In particular, different strategies of combining modules are used when transformations in an optimizer are performed. Transformation rules separated into several modules serve as a basis for transformation. Their combination by different above mentioned strategies allows to work with larger rule sets with special knowledge about combination included. These depend usually on the application (and database) domain.

5 Conclusions

Computer methods make use of both domain independent and dependent knowledge in one way or another. Writing such a knowledge explicitly is important for documentation purposes. Writing it at the same time in such a way that would allow direct processing is at least as important, having in mind method verification, comprehension, modification, maintenance etc. Declarative representation is one such way in general, and Prolog is a suitable language in particular. The experience has shown, however, that it lacks suitable structuring constructs to allow straightforward grouping of related knowledge pieces.

Our approach which is based on using multi-level logic programs allows to represent modules as well as relations among them by means of a uniform formalism (i.e., logic). In particular, the devised programming technique is suitable for design and experimental prototype of an optimizer which uses special knowledge. This knowledge is in most cases domain dependent.

The major contributions of our approach are: (i) proposal of a programming technique that allows to write optimizer in multiple hierarchical levels; (ii) representation of some optimization techniques and strategies in this framework, (iii) representation of knowledge about combining different modules [2].

Appendix

Inference rules used by an abstract machine to process a multilevel logic program

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 $mgu(A, A')$ denote the most general unifier of two atomic formulae A and A'. Let $mod(P)$ denote set of names of modules of program P and $|M|$ denote 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 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*).

References

[1] M.M. Astrahan, et. al. *System/R: a relational approach to database management.* ACM Transactions on Database Systems, 1(2):97-137, 1976.

[2] M. Bieliková, P. Návrat. *A Programming Technique of Implementing Multi-level Logic Programs.* Tech. Report, STU Bratislava, 1995.

[3] M. Bieliková, B. Finance, G. Gardarin, L. Molnár, P. Návrat, M. Smolárová, Z. Tang. *Modeling a Query Optimizer with Multi-Level Logic Programming.* Revue de ingenerie d'information, 5(2):195-218, 1997.

[4] A. Brogi, E. Lamma, P. Mello. *A general framework for structuring logic programs.* Tech. report, 4/1, CNR Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, May 1990.

[5] A. Brogi. *Program Construction in Computational Logic.* Ph.D. Thesis: TD-2/93, Universita di Pisa, March 1993.

[6] D. Chretien, F. Machuca, P. Om, Z.H. Tang. *Cost-Controlled OFL Rewriting Rules for Multiple Collection Traversals.* Copernicus internal report on query optimization, 1995.

[7] B. Finance. *A rule-based query optimizer with multiple search strategies.* Data & Knowledge Engineering, 13:1-19, 1994.

[8] D. Friedman, M. Wand. *Reification: Reflection without meta-physic.* In Proc. of ACM Symp. on LISP and Functional Programming, Austin, pages 348-355. 1984.

[9] G. Gardarin, F. Machuca, P. Pucheral. *OFL: A Functional Execution Model for Object Query Languages.* In Proc. of ACM SIGMOD Int. Conf., USA, pages 59-70. 1995.

[10] C. Galindo-Legaria, J. Pellenkoft, M.L. Kersten. *Cost distributions of search spaces in query optimization.* Report CS-R9432 CWI Amsterdam, July 1994.

[11] L. Giordano, A. Martelli, G. Rossi. *Structured Prolog: A Language for Structured Logic Programming.* Software – Concepts and Tools, 15, pages 125-145. Springer-Verlag 1994.

[12] G. Graefe, D. DeWitt. *The EXODUS Optimizer Generator.* Ph.D. Thesis, University of Wisconsin, August 1987 and published at ACM-SIGMOD Int. Conf., San Fransisco, 1987.

[13] L.M. Haas, J.C. Freytag, G.M. Lohman, H. Pirahesh. *Extensible Query Processing in Starburst.* IBM Research Report RJ 6610 (63921),1988.

[14] Y.E. Ioannidis, Y. Lashkari. *Incomplete Path Expressions and their Disambiguation.* ACM SIGMOD, 23(2):138-149, 1994.

[15] R.A. Kowalski. *Logic for problem solving.* North-Holland, 1979.

[16] E. Lamma, P. Mello, A. Natali. *An extended Warren abstract machine for the execution of structured logic programs.* J. of Logic Programming, 14:187-222, 1992.

[17] L. Leonardi, P. Mello, A. Natali. *Prototypes in Prolog.* J. of Object-Oriented Programming, 2(3):20-28, 1989.

[18] F.G. McCabe. *Logic and Objects.* Prentice Hall, 1992.

[19] P. Mello. *Inheritance as Combination of Horn Clause Theories.* In Inheritance Hierarchies in Knowledge Representation and Programming Languages, M. Lenzerini, D. Nardi, M. Simi (Eds.), John Wiley & Sons Ltd, pages 275-289. 1991.

[20] L. Monteiro, A. Porto. *Contextual Logic Programming.* In Proc. 6th Int. Conf. and Symposium on Logic Programming. G. Levi, M. Martelli (Eds.), The MIT Press, Cambridge (USA), pages 1-26. 1989.

[21] D.T. Sannella, L.A. Wallen. *A calculus for the construction of modular prolog programs.* J. of Logic Programming, 12:147-177, 1992.

[22] E. Sciore, J. Sieg. *A Modular Optimizer Generator.* IEEE Transactions on Knowledge and Data Engineering, 2(1), 1990.

[23] P. Selinger et. al. *Access Path Selection in a Relational Database Management System.* In Proc. of the ACM SIGMOD Conf., New York, 1979.

[24] L. Sterling, R.D. Beer. *Metainterpreters for expert system construction.* J. of Logic Programming, 147-177, 1989.

[25] M. Stonebraker, E. Wong, P. Kreps, G. Held. *The Design and Implementation of INGRES.* ACM Trans. on Database Systems, 1 (3), 1976.

[26] D. Xu, G.Zheng. *Logical Objects with Constraints.* ACM SIGPLAN Notices, 30(1), pages 5-10. 1995.