Modeling a Query Optimizer with Multi-Level Logic Programming

Mária Bieliková[a], Béatrice Finance[b], Georges Gardarin[b], Ľudovít Molnár[a], Pavol Návrat[a], Mária Smolárová[a], Zhao-Hui Tang[b]

[a]*Slovak Technical University*
Dept. of Comp. Sci. and Eng.
Ilkovičova 3,
812 19 Bratislava, Slovakia
lastname@*elf.stuba.sk*

[b]*CNRS-PriSM Laboratory*
Univ. of Versailles-St-Quentin
78035 Versailles, France
firstname.lastname@*prism.uvsq.fr*

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 both query rewriting and planning, as well as search strategies. 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. This research has been motivated by the needs to provide more lucid and effective means for describing and structuring all the various knowledge contained in a query optimizer. To illustrate these ideas, we show how multi-level programming can be used to model a query optimizer for an object-oriented database.

KEY WORDS: query optimization, object-oriented database, module, meta-interpretation, multi-level logic programming, Prolog

---

1. Introduction

Query optimization in extended relational, object-oriented and deductive systems is a key issue in current database. 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 [HAA88]: *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 (i.e., INGRES [Stonebraker76], SYSTEM/R [AST76]) followed a 'black-box' approach. The optimizer knowledge was procedural making it difficult to evolve. Recently, extensible optimizers have been proposed (i.e., EXODUS [Graefe87], STARBURST [HAS88], [FRE87, SCI90]). The key idea was to generate a query optimizer from rules for transforming plans into alternative plans. Many rules languages have been proposed : term rewriting, functional or algebra equivalences (see [FIN92] for a survey). To control the rule evaluation, many search strategies have been also proposed including variations of enumerative search [SEL79] and randomized search [IOA87, Ioannidis90, Swammi89]. 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. The idea is to apply some technics already proposed in logic programming. Indeed, they have already noticed that, despite several widely recognised advantages of logic programming (i.e., declarative semantics, or mechanisms of unification and deduction), the extent of its suitability for development of large systems is relatively limited. One of the crucial problems 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. It is particulary true in query optimization, were methods and technics for query optimization are based on various kinds of knowledge. Among them, we find : (1) technics 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 of the COPERNICUS project. This query optimizer has been specified by the PRiSM Laboratory [CHR95]. Optimization rules are described by using OFL (Object Functional Language) [GAB95]. OFL is a target object language for OQL-like [CAT93] query compilers introduced in [GAA95]. 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 following properties regarding the optimization : (1) recognition of hierarchical levels within the relevant knowledge, (2) structuralization of optimization knowledge which enables better understanding, readability and maintenance, (3) extensibility of the optimizer (possibility to improve optimizer as the field evolves), (4) easy combining of different methods and techniques by specifying alternative modules and meta-modules which control their use in the optimization process, and (5) development and experimental testing of new approaches (e.g. new transformation rules and strategies which are domain dependent).

The paper is organized as follows : in Section 2, we an overview of a programming technique for implementing multi-level logic programs. In Section 3, we describe the architecture of the optimizer in terms of modules and multi-levels. In Section 4, we define each module by a multi-level program and we show how modules can be combined together. Then, we conclude.

2. A Programming Technique for Implementing Multi-level Logic Programs

As we mentioned already above, our research has been motivated by the need to extend and conveniently represent new approaches to query optimization. We concentrate on the way how to represent different approaches in uniform (logic) formalism and how to combine them. A logic formalism, like any declarative formalism in general, is an excellent tool to express algorithmic knowledge at a very high level of abstraction. It is convenient for the developer because it aids to reduce the descriptive complexity, it supports rapid prototyping, it makes future modifications easier etc. We have chosen Prolog as a particular logic language because it is wide enough spread and used to provide substantial evidence of suitability in these kind of tasks. The choice of Prolog is further supported by the possibility of metaprogramming.

However, 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. Separated logic databases are called modules [CLA84, SAN92, MEI93, KWO93, GIO94], worlds, theories [MCC92], units [LAM92]. Several authors have applied concepts of the object oriented programming to achieve structuring of logic programs [KOW79, MEL91, MCC92, XUZ95].

Problems are encountered when trying to combine logic databases (modules). Several approaches have been tried, e.g. inheritance [MEL91], context switching [LAM92], introducing implication into goals [KWO93, GIO94], different definitions of visibility of atoms [MCC92], using abstraction in separating the logic database from the concrete implementation by specifying required resources and produced results [SAN92]. 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 [BRO93]. Some aspects of the logic program processing machine, not visible at the level of a standard logic program, can be made visible and accessible as first class objects [Cavalieri89].

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 [STE89]. Often, the chosen level of detail is the goal reduction level, with the program being represented at the meta-level by a predicate *clause/2* where the first parameter is clause head, and second parameter is clause body [LAM92, BRO93, DEN93]. However, it is possible to define meta-interpreters with a finer level of granularity, for instance one modelling also backtrack, or one explicitly implementing unification.

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. [FRI84], 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.

We shall present our proposal on how to extend a logic program with a possibility of representing parts of it at several levels. Our goal is to use such a multilevel logic program to represent object-oriented query optimizer. Our approach is based on the reflection technique as elaborated by Lamma, Mello, and Natali [LAM92] who used reflection for combining Prolog databases through contexts and inheritance. We report on a prototype implementation of the multilevel logic programming for Prolog. The implementation is meta-interpreted.

2.1. Reflection

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) [LAM92].

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 *(CM,CG,AUX)*, where *CM* is the current module, *CG* 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. 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([CM,CG,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([CM1,CG1,AUX1])*. This causes an object level computation to start in the module *CM1* aiming to prove the goal *CG1*. 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.

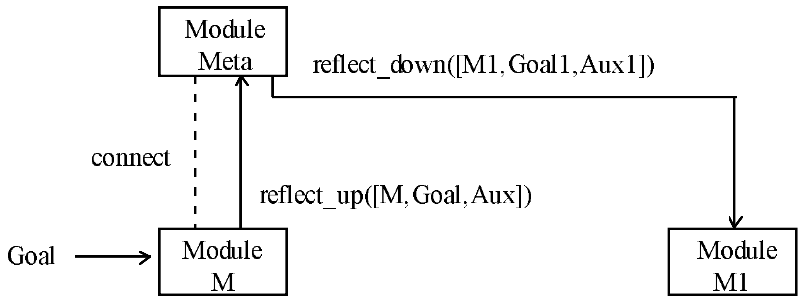

Figure 1. *Proof of a goal in the module M*

The above described reflection mechanism can be used in a simplest case for implementing a call between two modules. The reflection technique, involving the visible state of the abstract machine, allows to tackle various problems such as:

– describing various ways of combining modules,
– modifying dynamically the strategy of selecting modules for interpretation by changing the connection to meta-module.

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.

2.2. Design of a Multilevel Logic Program

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. In this section, we shall present our approach to defining such multi-level logic programs.

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 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.

In the appendix, we present inference rules which are used by the abstract machine to process a multi-level logic program. They serve as a basis for implementing a multi-level Prolog program interpreter. The proof at a meta-level has the same procedural semantics as at the program level. If a module M is *connected* to some other modules, upward reflection occurs to the next higher level. In particular, upward reflection can occur during an attempt to satisfy a goal *reflect_down*, too. However, if there is not *connected* any module to a given module during an attempt to satisfy a goal *reflect_down*, reflection occurs towards level determined by a parameter of the term *reflect_down*.

3. Query Optimizer Architecture

In this section, we try to identify the main components (modules) of a query optimizer.

Query optimization can be divided in two phases: *query rewriting* transforms queries into equivalent simpler ones with better expected performance and *query planning* primarily determines the method for accessing objects. In this paper, we concentrate on query rewriting technique. Query rewriting includes syntactic transformations such as query modification with views, factorization of common subexpressions, select migration through join, ordering the conditions in a sequence according to their relative selectivity and evaluation cost [CLU92, LAN92, PIR92, KEM94]. It also includes semantic transformations such as query simplification using integrity constraints [SIE92, SUN94], or knowledge about abstract data types [MIT91].

The goal of rewriting is to produce another query with better expected performance, which is the result of a transformation process of the initial query. An optimizer often works with some internal representation of a query. In our paper, OFL [GAA95] is used to represent queries.

Rule rewriting take place in a search space with a defined search strategy. The search space is characterized by a set of operators, i.e., transformation rules along with their respective scopes of applicability. An action in the optimizer corresponds to a transformation rule application. The search strategy is responsible for controlling the application of such rules.

Usually, there are many alternatives from which to choose. Moreover, there can be many alternative expressions equivalent to the original query expression (OFL program). For example, each operator can have several alternative implementations; it can exist several access paths to the data [Mitchell92]. The optimizer explores the search space of alternatives using search (control) strategy together with a defined measure for evaluating alternatives (e.g. cost model).

In order to perform better rewriting, in particular to reduce the overall complexity, the optimizer should be able: (1) to focus on subproblems of different granularities, i.e. to vary subproblem granularity during optimization, (2) to work with different definitions of evaluations of queries and with different ways of incorporating them into the optimization strategy, (3) to provide more than a single strategy for transforming queries, (4) to use heuristics to limit the search space, and (5) to apply alternative equivalences between two queries to find better solutions (it can be convenient to determine when such transformations are acceptable).

There is an evident advantage of using rule paradigm for specifying the optimizer actions [Mitchell92]. Rule based paradigm has made it easy to exploit the complicated triggering interactions between rewriting rules. Moreover, it is an excellent platform for extensibility. Similar to the use of rule paradigm for knowledge represenation in artificial intelligence field, rules can be grouped into modules (rule sets) each of which can be considered separately. This gives a number of advantages: (1) declarativeness of the representations, (2) extensibility, (3) it allows to group rules into units of related rules, allowing for better comprehension of rule's operation, (4) different rule sets can have different properties defined, for example different control strategies (i.e., the way of interpretation), and (5) a possibility to use the same rule paradigm to define control (meta-rules).

When attempting to represent the optimizer that performs query rewriting, we first identify basic elements of the architecture, cf. [FIN94, Mitchell92]. Our approach offers sufficient means of representing multi-level architectures. Being based on a declarative formalism (i.e., a logic programming language), it is sufficiently abstract to allow to concentrate on essential features of the architecture. We have enhanced the language not only with constructs allowing to structure the descriptions into modules, but more importantly also with techniques of defining various ways of combining modules. Our approach allows to define generic search strategies that can be specialized according to domain related or other requirements.

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**.

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.

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 [Selinger87] and randomized search [Ionnidis94, SWA89, GAL94]. 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 [GAL94] are represented by a special module.

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 [BRO90, MEL91]. We were able to express them within our framework and we present them in section 4 because we feel they can be useful when combining knoweldge modules of a query optimizer, too. 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.

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

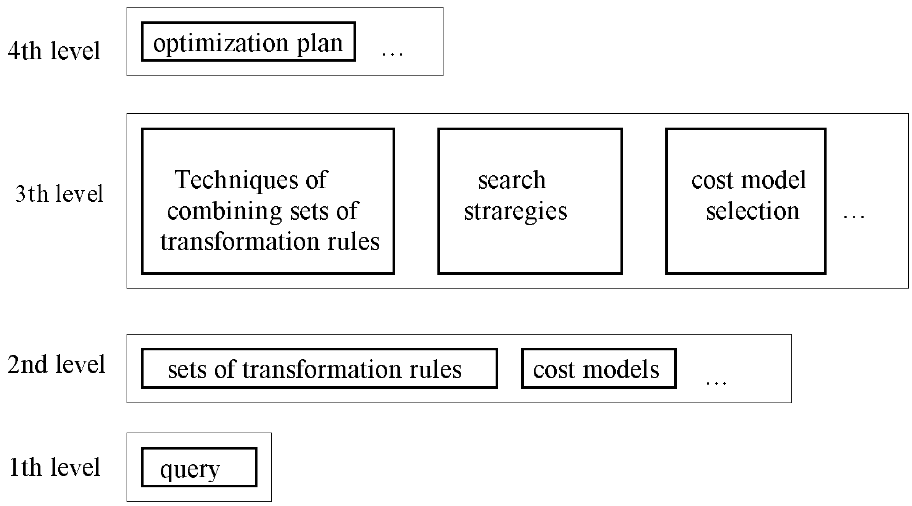

Figure 2 : *Optimizer Architecture*

Our approach which is based on using multilevel logic programs allows to represent modules as well as relations among them by means of a uniform formalism (i.e., logic). In the next section, we introduce basic ideas of multilevel logic (Prolog) programs which are used to express parts of the optimizer.

4. Optimizer Representation by a Multi-level Logic Program

Here we are going to present an application of the proposed programming technique. Multi-level Prolog programming can be used with advantage whenever 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 the object level knowledge should. Knowledge is better captured, understood, manipulated, and applied if it is structured into interconnected units. But supposing e.g. there are several problem solving methods defined at the meta-level, it is very likely that also another kind of knowledge is available, via 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 syntax oriented approaches.

For purposes of optimizer representation examples, modules belonging to the optimizer architecture defined above are briefly described. Their representation in multi-level Prolog is illustrated.

4.1. Modules

At the first level of the module hierarchy, there is represented a query. Query optimizers and optimization techniques are usually based on a particular data model and query language for that model. This paper is based on optimizing queries specified in OFL [GAA95]. The process of generating OFL programs from OQL like queries, and the implementation of the language on top of an object manager are desribed in [GAA95]. OFL supports any complex object algebra with recursion as macros. It manipulates abstract collections that model collections in the execution world of an OQL-like query compiler, namely class extents, multivalued attributes as well as indexes or hashing tables. Since abstract collections are physical entities OFL transformations can be cost-controlled [VAL91].

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 i.e., conditional and the quantifier functions forAll, forAny. Their parameters are written as facts.

To illustrate let us assume the functional ODL-like schema taken from [GAA95] presented in the following figure:

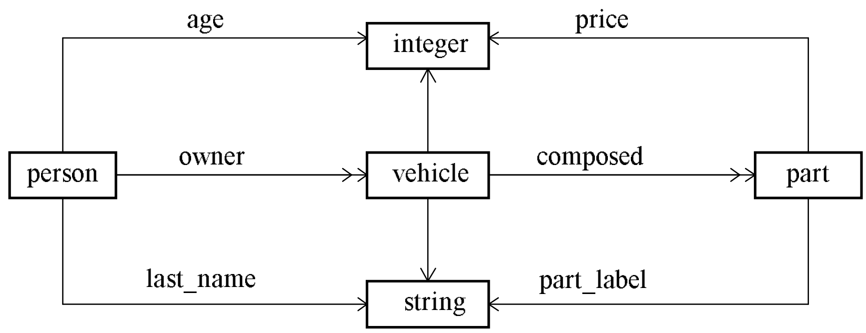

Figure 3. *Functional ODL-like schema example*

This ODL schema describes *persons* owning *vehicles* composed of *parts*; *last_name, color* and *part_label* are string valued attributes; *age* and *price* are integer valued attributes; *owner* and *composed* are set-valued attributes represented by double arrows in the Figure 3. We illustrate a possible representation of the following OFL query. For clarity the OQL query is presented first. Query selects the names of all 16 years old persons. We named **person'** the abstract collection representing person class extents.

OQL:	***OFL:***
SELECT p.last_name	**forAll**(person',
FROM p in Person	equal(age(current(person')), 16),
WHERE p.age = 16	display(last_name(current(person'))))

LOGIC FACTS :

query1 ismod

{ query(forAll(collection(1,C), condition(1,P), function(1,F))).

collection(1, person').
condition(1, equal(age(current(person')), 16)).
function(1, display(last_name(current(person')))) }.

We use operator *M* ismod *D* to assign name *M* to a set of clauses *D.* The first argument of the relations *collection, condition* and *function* serves as an identifier for determining the place where particular subexpression belongs. It can be generated automatically. Therefore translation of term representing OFL query to Prolog facts is straightforward.

At the next level, there are represented transformation rules. They can be grouped (structured) into subsets which are represented by modules. For example, we can group together rules used in transforming certain kind of subexpressions. Traditionally, a term rewrite rule is an expression $\tau_1 \rightarrow \tau_2$, where all variables in τ_2 also occur in τ_1 [DER87]. Such traditional term rewrite rules are not powerful enough to express complex query transformations, as required in a query optimizer.

New approaches focus not only on the structure properties of terms. They interpret the semantics of queries [FIN94].

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 represenation such as constraints, methods, differential cost computations.

The proposed representation of a rewrite rule is the following:

rule(if(IF-Term), then(THEN-Term),
List_of_additional_characteristics)

The rule is interpreted as:

> *"if IF-Term appears in the selected query it is rewritten as the given THEN-term only if List of additional characteristics is succesfully interpreted"*

The list of additional characteristics is left open and their interpretation can be defined in a separate module.

Next, we show examples of syntactically correct rules taken from [FIN94].

```
predicate_simplification ismod

{ %Rule 1
        rule(if(and(X, not(X))), then(false), []).

  %Rule 2
        rule(if(and(false, X)), then(false), []).

  %Rule 3
        rule(if(or(true, X)), then(true), [])  ....  }.
```

At the next level, search strategies controlling the application of transformation rules are represented. Besides defining transformation rules, a search strategy is, one of the most crucial component of an optimizer based on query rewriting technique. The space of semantically equivalent alternatives usually grows very quickly. There are well-known approaches to explore a search space. Deterministic search algorithms take exponential time on the number of collections of the query. Better candidates are probabilistic algorithms [FIN94, GAL94] such as simulated annealing, iterative improvement or transformation-free algorithm.

Each of these strategies can be represented in a separate module by multilevel Prolog. For illustration we show 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 <u>ismod</u> %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 a 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 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.

There are some strategies which do not fit into general pattern defined in module *search_strategy*. We show an example of one such strategy called transformation free strategy, which is presented in [GAL94].

transformation_free <u>ismod</u>

```
{   transf_free([Stop_Cond | _], Min_State, Min_State) :-

        call(Stop_Cond),!.
    transf_free([Stop_Cond | Optimiz_Par], Current_State, Min_State) :-
        get_random_state(New_State),
        is_better(Optimiz_Par, New_State, Current_State),!,
```

```
(   transf_free([Stop_Cond | Optimiz_Par],New_State, Min_State)
    ;
    transf_free([Stop_Cond | Optimiz_Par],Current_State, Min_State) )
    }.
```

In application of the above represented strategies, several decisions must be made. For example, how to generate a random state, how and how many neighbors to generate, which neighbor to select for exploration, the way of comparing two states. A suitable place for writing this knowledge is at several levels.

In the case of selecting a neighbor for exploration, the simplest way is to select it according to the order in which the neighbors are generated. The method for selection would be in that case implemented by the predicate *member/2*. Other methods are also possible, such as random selection or selection according to an estimated cost. The method for selection is defined at a meta-level. In such a way, different methods can be applied according to actual requirement simply by switching connection between the level of strategy representation and the meta-level of method for selection. The latter level can be defined as follows:

```
meta_order_selection ismod
  { reflect_up([_, select_neighbor(List, Element), AUX]) :-
        reflect_down([methods_of_selection, member(Element, List), AUX])
        }.

meta_random_selection ismod
  { reflect_up([_, select_neighbor(List, Element), AUX]) :-
        reflect_down([methods_of_selection, random(List, Element), AUX])
        }.

methods_of_selection ismod
  { member(X, [X|_]).                          %order selection
    member(X, [_|R]) :- member(X, R).

    random(List, Element) :- ....              %random selection     }.
```

We present the module for selection according to order (module *meta_order_selection*) and the module for random selection (module *meta_random_selection*). Both modules use predicates defined in the module *methods_of_selection.* In this case, the relation between the module, *meta_order_selection,* and the module, *meta_random_selection,* represents simple procedure call. The *simulated_annealing* module is connected with one of these modules by the relation *connect.* When attempting to satisfy a goal, there occurs a reflection and a goal with the functor, *reflect_up* is proved as described in the Figure 1. In the above case, there occurs a reflection upwards to the level where knowledge on how to select is written, i.e. to the *meta_random_selection* module.

Now let us turn our attention to the way of implementing a comparison between the different states. For object-oriented queries, the need for cost-based optimization is evident [LAN92]. For experimental purposes, we can have several cost models, which may depend on the 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 the acceptance of the state. In the transformation-free algorithm, the state with 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 relation *is_better*.

An attempt to satisfy a goal *is_better(Optimiz_Par, State1, State2)* always causes reflection to a meta-level represented by a 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 *transformation_free* and *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 representating other concepts in optimization, including transformation rule representation (each set of rules in one module), cost model definition, method for random state generation, ...

At the highest level, the global optimization strategy is represented. This module can use for example a control strategy similar to ones used in space searching by artificial intelligence techniques such as hill climbing. Here a decision can be made about the 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 transformation free technique. Finally select the best solution.

4.2. 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*. It 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 as reported in [BNA95]. Now we briefly describe our main ideas.

• **Global strategy**

The combination of modules is the clause union of the modules being combined [Miller89]. The formal semantics of the operator '*' which combines two modules and which is commutative are defined in [BRO93].

• **Local strategy**

The combination of modules allows to infer only those formulae which can be inferred from single modules [GIO90]. Predicate definitions in modules are considered local. Modules are 'closed' [BRO90]. 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 the 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 [GIO94]. They are inspired from conventional programming languages. Moreover, we can mention different ways of inheritance [MEL91]. 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 [MON89]. A context is an ordered set of modules which can change during the process of proving a formula. On the 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. It 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**

The simple inheritance can be expressed as a special case of contextual programming. The context is just the 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 [LEO89]:

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

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:

– proposal of a programming technique that allows to write optimizer in multiple hierarchical levels
– representation of some optimization techniques and strategies in this framework
– representation of knowledge about combining different modules.

References

[AST76] Astrahan M.M, and al., "System/R: a relational approach to database management",, ACM Transactions on Database Systems 1(2), June, 1976, pp 97-137

[BLA93] J.A. Blakeley, W.J. McKenna, G. Graefe, "Experiences Building the Open OODB Query Optimizer", in ACM SIGMOD, Vol.22, Num.2, 1993, pp.287-296

[BNA95] M. Bieliková, P. Návrat, "A Programming Technique of Implementing Multilevel Logic Programs", Tech. Report, STU Bratislava, 1995

[BNA96] M. Bieliková, P. Návrat, "Knowledge Based Method for Building a Software System Configuration", Knowledge Based Systems, 9(1996), 1, pp. 61-65.

[BRO90] 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

[BRO93] A. Brogi, "Program Construction in Computational Logic", Ph.D. Thesis: TD-2/93, Universita di Pisa, March 1993

[CAV89] M. Cavalieri, E. Lamma, P. Mello, A. Natali, "Meta- Programming in Prolog Through Direct Introspection: A Comparison with Meta- Interpretation Techniques", in Proc. Meta- Programming in Logic Programming, H. Abramson, M. Rogers (Eds.), Massachusetts London, 1989, pp. 399-415

[CAT93] R.G.G. Cattel, Ed., "Object Databases: The ODMG-93 Standard", Book, Morgan & Kaufman, 1993

[CLA84] K.L. Clark, F.G. McCabe, "Micro-Prolog: Programming in Logic", International Series in Comp. Science, Prentice-Hall Int., 1984

[CLU92] S. Cluet, C. Delobel, "A General Framework for the Optimization of Object-Oriented Queries", in ACM SIGMOD, Vol.21, Issue 2, 1992, pp.383-392

[COL94] R.L. Cole, G. Graefe, "Optimization of Dynamic Query Evaluation Plans", in ACM SIGMOD, Vol.23, Num.2, 1994, pp.150-160

[CHR95] 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

[DEN93] E. Denti, E. Lamma, P. Mello, A. Natali, A. Omocini, "Techniques for Implementing Contexts in Logic Programming", in Extensions of Logic Programming, LNAI 660, E. Lamma, P. Mello (Eds.), Springer-Verlag 1993, pp.339-358

[DER87] N. Dershowitz, J.P. Jouannaud, "Rewriting Systems", in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Sciences, North-Holland, Amsterdam, 1987

[FIN92] B. Finance, "Une plate-forme pour la génération d'optimiseurs extensibles", Ph.D. Thesis, University of Paris 6, 1992.
[FIN94] B. Finance, "A rule-based query optimizer with multiple search strategies", in Data & Knowledge Engineering 13, 1994, pp.1-19
[FRE87] Freytag J.C., "A Rule-Based View of Query Optimization", ACM SIGMOD Int. Conf., San Fransisco, 1987
[FRI84] D. Friedman, M. Wand: "Reification: Reflection without meta-physic", in Proc. of ACM Symp. on LISP and Functional Programming, Austin, August 1984, pp.348-355
[GAA95] G. Gardarin, F. Machuca, P. Pucheral, "OFL: A Functional Execution Model for Object Query Languages", in Proc. of ACM SIGMOD Int. Conf., USA, May 1995, pp.59-70
[GAB95] G. Gardarin, J.R. Gruser, Z.H. Tang, "Efficient Processing of Path Expressions", Tech. Report, PRiSM Laboratory, France, 1995
[GAL94] C. Galindo-Legaria, J. Pellenkoft, M.L. Kersten, "Cost distributions of search spaces in query optimization", Report CS-R9432 CWI Amsterdam, July 1994
[GAR94] G. Gardarin, F. Machuca, "Query Optimization as Object Functional Language (OFL) Transformations", Technical Report, PRISM Laboratory, 1994
[GIO90] L. Giordano, A. Martelli, G.V. Rossi, "Extending Horn clause logic with module constructs", Tech. Rep. UDMI/04/90/RR, Univ. Udine, Italy, 1990, 18 pages
[GIO94] L. Giordano, A. Martelli, G. Rossi, "Structured Prolog: A Language for Structured Logic Programming", in Software - Concepts and Tools, 15, Springer-Verlag 1994, pp.125-145
[GRA87] Graefe G. and DeWitt D., "The EXODUS Optimizer Generator", Ph.D. Thesis, University of Wisconsin, August 1987 and published at ACM-SIGMOD Int. Conf., San Fransisco, 1987
[HAA88] Haas L.M. and Freytag J.C. and Lohman G.M. and Pirahesh H., "Extensible Query Processing in Starburst", IBM Research Report RJ 6610 (63921),1988.
[HAS88] Hasan W. and Pirahesh H., "Query Rewrite Optimization in Starburst", IBM Research Report RJ 6367 (62349),1988
[IOA87] Ioannidis Y. and Wong E., "Query Optimization by Simulated Annealing", ACM SIGMOD Int. Conf., San Fransisco, CA, 1987.
[IOA94] Y.E. Ioannidis, Y. Lashkari, "Incomplete Path Expressions and their Disambiguation", in ACM SIGMOD, Vol.23, Num.2, 1994, pp.138-149
[KEM94] A. Kemper, G. Moerkotte, K. Peithner, M. Steinbrunn, "Optimizing Disjunctive Queries with Expensive Predicates", in ACM SIGMOD, Vol.23, Num.2, 1994, pp.336-347
[KOW79] R.A. Kowalski, "Logic for problem solving", North-Holland, 1979
[KWO93] K. Keehang, G. Nadathur, D.S. Wilson, "Implementing a Notion of Modules in the Logic Programming Language ?Prolog", in Extensions of Logic Programming, LNAI 660, E. Lamma, P. Mello (Eds.), Springer-Verlag 1993, pp.359-393
[LAM92] E. Lamma, P. Mello, A. Natali, "An extended Warren abstract machine for the execution of structured logic programs", in J. Logic programming, No.14, 1992, pp.187-222
[LAN92] R.S.G. Lanzelotte, P. Valduries, M. Zait, "Optimization of Object-Oriented Recursive Queries using Cost-Controlled Strategies", in ACM SIGMOD, Vol.21, Issue 2, 1992, pp.2566-265
[LEO89] L. Leonardi, P. Mello, A. Natali, "Prototypes in Prolog", in Journal of Object-Oriented programming", Vol.2, No.3, Sept/Oct 1989, pp.20-28
[MCC92] F.G. McCabe, "Logic and Objects", Prentice Hall, 1992

[MEI93] M. Meier, J. Schimpf, "An Architecture for Prolog Extensions", in Extensions of Logic Programming, LNAI 660, E. Lamma, P. Mello (Eds.), Springer-Verlag 1993, pp.319-338

[MEL91] 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 1991, pp.275-289

[MIL89] D. Miller, "A logical analysis of modules in logic programming", in Journal of Logic Programming, Vol.6, 1989, pp.79-108

[MIT91] G. Mitchell, S.B. Zdonik, U. Dayal, "Object-Oriented Query Optimization: What's the Problem?", Tech. Report No. CS-91-41, Brown University, June 1991

[Mitchell92] G. Mitchell, S.B. Zdonik, U. Dayal, "An Architecture for Query Processing in Persistent Object Stores", in Proc. Hawai Conf. on System Sciences, 1992, 17 pages

[MON89] 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), 1989, pp.1-26

[PIR92] H. Pirahesh, J.M. Hellerstein, W. Hasan, "Extensible/Rule Based Query Rewrite Optimization in Starburst", in ACM SIGMOD, Vol.2, Iss.2, 1992, pp.39-48

[SAN92] D.T. Sannella, L.A. Wallen, "A calculus for the construction of modular prolog programs", in The Journal of logic programming, No. 12, 1992, pp.147-177

[SCI90] Sciore E. and Sieg J., A Modular Optimizer Generator, IEEE Transactions on Knowledge and Data Engineering, 2(1), 1990

[SEL79] P. Selinger et. al., "Access Path Selection in a Relational Database Management System", in Proc. of the ACM SIGMOD Conf., New York, 1979

[SHE89] P.C. Sheu, R.L. Kashayp, S. Yoo, "Query optimization in object-oriented knowledge bases", in Data & Knowledge Engineering 3, 1989, North-Holland, pp.285-302

[SIE92] M. Siegel, E. Sciore, S. Salveter, "A Method for Automatic Rule Derivation to Support Semantic Query Optimization", in ACM Trans. on Database Systems, Vol.17, No.4, Dec.1992, pp.563-600

[STE89] L. Sterling, R.D. Beer, "Metainterpreters for expert system construction", in J. Logic Programming, 1989, pp.163-178

[STO86] Stonebraker M., Wong E., Kreps P. and Held G., The Design and Implementation of INGRES, ACM Trans. on Database Systems, 1 (3), 1976

[SUN94] W. Sun, C.T. Yu, "Semantic Query Optimization for Tree and Chain Queries", in IEEE Trans. on Knowledge and Data Engineering, Vol.6, No.1, February 1994, pp.136-151

[SWA89] A. Swami, "Optimization of large join queries: Combining heuristics and combinatorial techniques", in ACM SIGMOD Int. Conf., Portland, 1989

[VAL91] P. Valduriez, R. Lanzelotte, M. Ziane, JP. Cheiney, "Optimization of non Recursive Queries in OODB", In Proc. DOOD, Munich, Germany, 1991

[XUZ95] D.Xu, G.Zheng, "Logical Objects with Constraints", in ACM SIGPLAN Notices, Vol.30, No.1, January 1995, pp.5-10