Using Queries with Multi-Directional Functions for Numerical Database Applications

Staffan Flodin, Kjell Orsborn, and Tore Risch

Department of Computer and Information Science
Linköping University, Sweden
stafl@ida.liu.se, kjeor@ida.liu.se, torri@ida.liu.se

Presented at 2nd East-European Symposium on Advances in Databases and Information Systems (ADBIS'98), September 1998

Abstract. Object-oriented database management systems are often motivated by their support for new emerging application areas such as computer-aided design and analysis systems. The object-oriented data model is well suited for managing the data complexity and representation needs of such applications. We have built a system for finite element analysis using an object-relational database management system. Our application domain needs customized numerical data representations and an object-oriented data model extended with multi-methods where several arguments are used in type resolution. To efficiently process queries involving domain functions in this environment without having to copy data to the application, support is needed for queries calling multi-methods with any configuration of bound or unbound arguments, called multi-directional functions. We show how to model multi-directional functions views containing matrix algebra operators, how to process queries to these views, and how to execute these queries in presence of late binding.

1 Introduction

Object-oriented database management systems (OODBMSs) [8][3][34] are often motivated by applications such as systems for computer-aided design and analysis [4][5][19]. The object-oriented data model is well suited for managing the complexity of the data in such applications. We have built a system for finite-element analysis (FEA) storing numerical data in an object-relational DBMS [25][26][27]. The performance requirements on the numerical data access in an FEA model are very high, and special methods have been developed for efficient representation of, e.g., matrices used in FEA. For instance, a fundamental component in an FEA system is a equation solving sub-system where one is interested in solving linear equation

systems such as $\mathbf{K} \times \mathbf{a} = \mathbf{f}$ where **a** is sought while **K** and **f** are known. Special representation methods and special equation solving methods must be applied that are dependent on the properties of the matrices. To get good performance in queries involving domain operators it is desirable to store both domain-oriented data representations and functions in the database. It is desirable to be able to evaluate the functions in the database server in order to avoid data shipping to the client. For this modern Object-Relational DBMS therefore provide User Defined Functions (UDFs) [20].

View in an object-oriented data model can be represented as functions (or methods) defined using queries containing operators from the domain, e.g. for matrix algebra [23]. Such *derived functions* expressed by side-effect free queries have a high abstraction level which is problem oriented and reusable. Query optimization techniques can be used to optimize such function definitions. To represent the model, an object-relational DBMS with extensible storage representation and query optimization is needed. The query optimizer needs to have special optimization methods and cost formulae for the FEA domain operators.

In pure object-oriented models the function (method) invocation is based on the *message-passing* paradigm where only the receiver of the message (the first argument of a function) participate in *type resolution*, i.e. selecting which *resolvent* (variant of an overloaded method or function name) to apply. The message-passing style of method invocation restricts the way in which relations between objects can be expressed [2].

In order to model a computational domain such as the FEA domain it is desirable to also support *multi-methods* [2] corresponding to functions with more than one argument. With multi-methods all arguments are used in type resolution. Thus, by extending the data model to incorporate multi-methods e.g. matrix operators applied on various kinds of matrix representations can be expressed in a natural way. In Sect. 4 we show how matrix operators are modeled as multi-methods in our application.

```
DECLARE mx1 AS matrix_type_1
DECLARE mx2 AS matrix_type_2
SELECT x FROM Matrix x
            WHERE mx1 * x = mx2 AND x IN f();
```

Fig. 1. A sample query exemplifying a multi-directional method

High level declarative queries with method calls do not specify exactly how a function is to be invoked in the query. We will show that *multi-directional functions* are needed in the database query language [15][16] for processing of queries involving inverses of functions. A function in the database query language is multi-directional if, for an arbitrary function invocation $m(x) = y$, it is possible to retrieve those arguments x that are mapped by the function m to a particular result, y. Multi-directional functions can furthermore have more than one argument. This ability provides the user with a declarative and flexible query language where the user does not have to specify explicitly how a function should be called. To exemplify a multi-directional function, consider the AMOSQL [17] query in Fig. 1 that retrieves those

matrices which, when multiplied by the matrix bound to the variable `mx1`, equals the matrix bound to the variable `mx2`.

This query is a declarative specification of the retrieval of the matrix `x` from the result set of the function `f()` where `x` solves the equation system `mx1 * x = mx2` (`mx1` and `mx2` are assumed given). It is the task of the query processor to find an efficient execution strategy for the declarative specification, e.g. by using the inverse of the `*` method (matrix multiplication) that first solves the equation system to get a value for `x`. An alternative execution strategy is to go through all matrixes `x` in *f(x)* and multiply them with `mx1` to compare the result with `mx2`. The first strategy is clearly better.

To provide a query optimizer with all possible execution plans it must be able to break encapsulation to expand the definitions of all referenced views [32]; thus the optimization is a *global optimization* method. In global optimization the implementations of all referenced views are substituted by their calls at *compile time*. For ORDBMS this means that the definitions of derived functions must be expanded before queries can be fully optimized, a process called *revelation* [12]. With revelation the query optimizer is allowed break encapsulation while the user still cannot access encapsulated data.

The object-oriented features include *inheritance*, *operator overloading*, *operator overriding* and *encapsulation* [3]. The combination of inheritance in the type hierarchy and operator overriding results in the requirement of having to select at run-time which resolvent to apply, i.e. *late binding*.

A function which is late bound obstructs global optimization since the resolvent cannot be selected until *run-time*. This may cause indexes and other properties that are important to achieve good performance to be hidden for the optimizer inside function definitions and remain unused during execution. Thus, late bound functions may cause severe performance degradation if not special query processing techniques are applied. This is why providing a solution that enables optimization of late bound functions is an important issue in the context of a database [12].

A special problem is the combination of late bound functions and multi-directional functions. This problem is addressed in [16] where late bound function calls are represented by a special algebra operator, `DTR`, in the execution plan. The `DTR` operator is defined in terms of the *possible resolvents*, i.e. the resolvents eligible for execution at run-time. Each resolvent is optimized with respect to the enclosing query plan. The cost model and selectivity prediction of the `DTR` operator is defined in terms of the costs and selectivities of the possible resolvents. The DTR-approach in [16] has been generalized to handle multi-methods [15].

In this paper we show, by using excerpts from our FEA application, that extending a pure object-oriented data model with multi-directional functions results in a system where complex applications can be modeled easily compared to modeling within a pure object-oriented data model. We furthermore show how such queries are translated into an algebraic representation for evaluation.

2 Related Work

Modeling matrix computations using the object-oriented paradigm has been addressed in e.g. [28][29]. In [21] an algebra for primitive matrix operations is proposed. None of those papers address late binding, multi-methods, or multi-directional functions. We will show the benefits of having these features when modeling complex applications.

OODBMSs have been the subject of extensive research during the last decade, e.g. [3][4][5][34]. Several OO query languages [9][24][23] have been proposed. However, to the best of our knowledge processing queries with multi-directional functions in OODBMSs has not yet been addressed.

Multi-methods require generalized type resolution methods compared to the type resolution of pure object-oriented methods [1][2]. In the database context good query optimization is another important issue.

Advanced applications, such as our FEA application, require domain dependent data representations of matrixes and an extensible object-relational query optimizer [33] to process queries over these representations. For accessing domain specific physical representations we have extended the technique of multi-directional foreign functions described in [22].

In [15][16] the optimization of queries with multi-directional late bound functions in a pure object-oriented model is addressed and the `DTR` operator is defined and proven to be efficient. In this paper that approach is generalized to multi-methods.

3 Background

In this section we first give a short introduction to the data model we use. Then properties of multi-directional functions are discussed followed by an overview of the issues related to late binding.

3.1 The Functional Data Model

The functional data model DAPLEX [31] has the notions of *entity* and *function*. A function is a mapping from entities to sets of entities. Based on DAPLEX the AMOS [13] data model contains *stored functions* and *derived functions*. Stored functions store properties of objects and correspond to attributes in the relational model and the object-oriented model. Derived functions are used to derive new properties which are not explicitly stored in the database. A derived function is defined by a query and corresponds to a view in the relational model and to a function (method) in the object-oriented model. In addition to stored and derived functions AMOS also has *foreign functions* which are defined using an auxiliary programming language such as C++ and then introduced into the query language [22]. In Fig. 1 the operator `*` is implemented by foreign functions. The only way to access properties of objects is through functions, thus functions provide *encapsulation* of the objects in the database.

3.2 Multi-Directional Functions

Multi-directional functions are functions which may be called with several different configurations of bound or unbound arguments and result, called *binding-patterns*. The query compiler must be capable of generating an optimal execution plan choosing among the possible binding-patterns for each multi-directional function. Sometimes such an execution plan may not exist and the query processor must then report the query as being unexecutable.

To denote which binding-pattern a function is called with, the arguments, a_i, and result, r, are annotated with b or f meaning bound or free as a^b_i or a^f_i if a_i is bound or free, respectively. In Fig. 1 the function * is called with its second argument unbound and with the first argument and the result bound. Thus, the call to * in that example will be denoted as $x^b \times y^f \rightarrow r^b$.

```
CREATE FUNCTION times(SymmetricMx x, ColumnMx y) -> ColumnMx r AS
                MULTIDIRECTIONAL
                  'bbf' FOREIGN MatrixMultiplication COST MultCost
                  'bfb' FOREIGN GaussDecomposition COST GaussCost;
```

Fig. 2. Creation of a multi-directional foreign function

Recall the three types of functions in the AMOS data model; stored, foreign and derived functions. Stored functions are made multi-directional by having the system automatically derive access plans for all binding-pattern configurations. Derived functions are made multi-directional by accessing their definitions and finding efficient execution plans for binding-patterns when needed. For multi-directional foreign functions the programmer has to explicitly assign each binding-pattern configuration an implementation, as illustrated in Fig. 2.

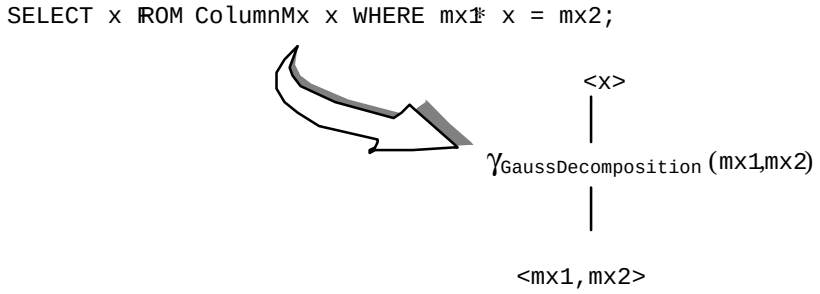

Fig. 3. Multi-directional function execution

In Fig. 2 the function `times` (implements `*`) is defined for two arguments and made multi-directional by defining which implementation to use for a certain binding-pattern. For $times(a^b,b^f) \rightarrow r^b$ the foreign function definition `GaussDecomposition` implements `times`, while `MatrixMultiplication` will be used for $times(a^b,b^b) \rightarrow r^f$. The functions `GaussDecomposition` and `MatrixMultiplication` are implemented in a conventional programming language such as C++. The implementor also provides optional cost functions, `MultCost` and

`GaussCost`, which are applied by the query optimizer to compute selectivities and costs.

The query processor translates the AMOSQL query into an internal algebra expression. Fig. 3 gives an example of a query with a multi-directional function and the corresponding algebra tree. Here the query interpreter must use $times(a^b,b^f)\rightarrow r^b$ which is a multi-directional foreign function (Fig. 2). The chosen implementation of `times`, i.e. `GaussDecomposition`, depends on the types of `mx1` and `mx2`. It will be called by the `apply` algebra operator, γ, which takes as input a tuple of objects and applies the subscripted function (here `GaussDecomposition`) to get the result.

Multi-directional functions enhance the expressive power of the data model but the query processor must include algorithms for type resolution of multi-methods [2] and for handling ambiguities [1].

3.3 Function Overloading and Late Binding

In the object-oriented data model [3] types are organized in a hierarchy with inheritance where subtypes inherit properties from their supertypes. *Overloading* is a feature which lets the same name denote several variants. For function names such variants are called *resolvents.* Resolvents are named by annotating the function name with the type of the arguments and result. The naming convention chosen in AMOS (and in this paper) is: `t1.t2.....tn.m -> tr` for a function m whose argument types are `t1,t2,...,tn` and result type is `tr`.

When names are overloaded within the transitive closure of a subtype-supertype relationship that name is said to be *overridden.*

In our object-oriented model an instance of type *t* is also an instance of all supertypes to that type, i.e. *inclusion polymorphism* [7]. Thus, any reference declared to denote objects of a particular type, *t*, may denote objects of type *t* or any subtype, t_{sub}, of that type. This is called *substitutability*. As a consequence of substitutability and overriding, functions may be required to be *late bound*.

For multi-directional functions the criteria for late binding is similar as for pure object-oriented methods [16] with the difference that for multi-directional functions, *tuple types* are considered instead of single types.

The query compiler resolves which function calls require late binding. Whenever late binding is required a special operator, `DTR`, is inserted into the calculus expression. The optimizer translates each `DTR` call into the special algebra operator γ_{DTR} which, among a set of possible resolvents, selects the subplan to execute according to the types of its arguments. This will be illustrated below. If the call does not require late binding it is either substituted by its body if a stored or derived function is called, or to a function application if a foreign function is called.

Special execution sub-plans are generated for the possible resolvents of the specific binding-patterns that are used in the execution plan. Thus available indexes will be utilized or other useful optimization will be performed on the possible resolvents. If any of the possible resolvents are unexecutable the enclosing `DTR` will also be unexecutable. The cost and selectivity of the `DTR` operator is calculated based on the costs and selectivities of the possible resolvents as their maximum cost and

minimum selectivity, respectively. Hence, `DTR` is used by a cost-based optimizer [30] to find an efficient execution strategy [16].

4 A Finite Element Analysis Example

A finite element analysis application has been modeled using the AMOS object-relational (OR) DBMS prototype. The purpose of this work is to investigate how a complex application can be built using the queries and views of an ORDBMS, and to develop suitable query processing algorithms.

4.1 Queries for Solving Linear Equations

A fundamental component in an FEA system is a linear equation solving sub-system. To model an equation solving system we define a type hierarchy of matrix types as illustrated in Fig. 4. Furthermore, the matrix multiplication operator, ×, is defined as functions on this matrix type hierarchy for several combinations of arguments. Fig. 5 illustrates how each variant of the multiplication function takes various matrix types (shapes) as arguments.

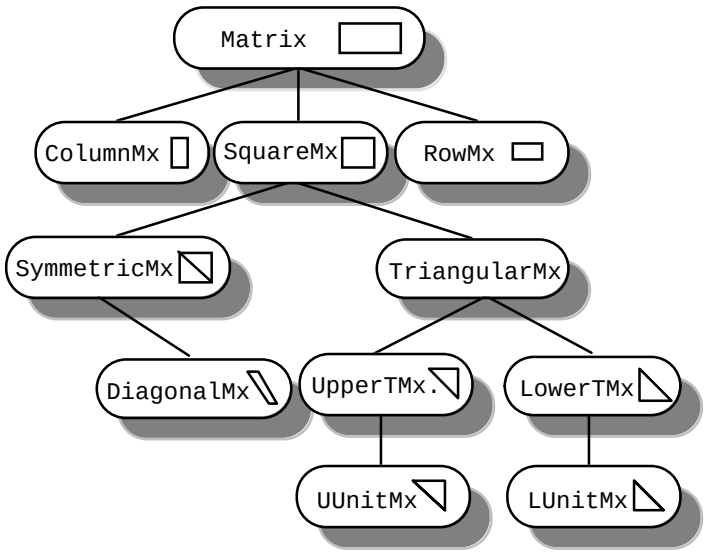

Fig. 4. A matrix type hierarchy for linear matrix algebra

The functions for multiplication are used to specify linear equation systems as matrix multiplications as **K** × **a** = **f**, where **a** is sought while **K** and **f** are known. Special representation methods and special equation solving methods have been developed that are dependent on the properties of the matrices. In our system, the function `times` (= infix operator `*`) is overloaded on both its arguments and has different implementations depending on the type (and thus representations) of the

matrices used in its arguments. For example, when **K** is a symmetric matrix, i.e. ⊠ × ▯ = ▯, it can be solved by a method that explores the symmetric properties of the first argument. One such method, $\mathbf{LDL}^T$ decomposition [18] illustrated in Fig. 6, substitutes the equation system with several equivalent equation systems that are simpler to solve.

The linear equation system is solved by starting with the factorization, $\mathbf{K} = \mathbf{U}^T \times \mathbf{D} \times \mathbf{U}$, that transforms *K* into the three matrices $\mathbf{U}^T$, **D**, and **U**. Then the upper triangular equation system $\mathbf{U}^T \times \mathbf{y} = \mathbf{f}$ is solved to get **y**. The diagonal equation system $\mathbf{D} \times \mathbf{x} = \mathbf{y}$ is then solved to get **x**, and finally the solution of the lower triangular equation system $\mathbf{U} \times \mathbf{a} = \mathbf{x}$ gets **a**. If the equation on the other hand is typed as □ × ▯ = ▯ some other method to solve it must be used, e.g. Gauss decomposition.

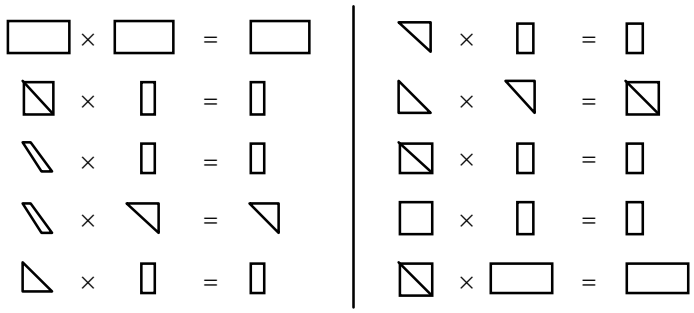

Fig. 5. A graphical notation for various matrix multiplication operators

The rationale for having these different overloaded matrix multiplication functions is efficiency and reusability. Efficiency because mathematical properties of the more specialized matrix types can be considered when implementing multiplication operators for them. Furthermore, specialized physical representations have been developed for many of the matrix types, e.g. to suppress zeroes or symmetric elements, and the matrix operators are defined in terms of these representations. The more specialized operators will in general have lower execution costs than the more general ones. The overloading provides reusability because every multiplication operator may be used in different contexts. To exemplify this consider the example in Fig. 7 over the type hierarchy in Fig. 4 and the multiplication operators from Fig. 5.

$$(\mathbf{K}^b \times \mathbf{a}^f = \mathbf{f}^b) \rightarrow \left\{ \begin{array}{rcl} \mathbf{K}^b & = & (\mathbf{U}^T)^f \times \mathbf{D}^f \times \mathbf{U}^f \\ \mathbf{U}^b \times \mathbf{a}^f & = & \mathbf{x}^b \\ \mathbf{D}^b \times \mathbf{x}^f & = & \mathbf{y}^b \\ (\mathbf{U}^T)^b \times \mathbf{y}^f & = & \mathbf{f}^b \end{array} \right\}$$

Fig. 6. The rewrite of a matrix multiplication expression to solve the equation system using **LDL**^{T} decomposition.

In this example a query is stated that solves an equation system by taking one square and one column matrix as arguments and calculating the solution by using the multiplication function. Depending on the type of the arguments the appropriate function from the type hierarchy below `SquareMx` will be selected at run-time, i.e. late binding. Also note that here the multiplication function (`*`) is used with the first argument and the result bound and the second argument unbound. This is only possible when multi-directional functions are supported by the system.

With multi-directional functions the equation system in Fig. 7 can be written as **K** × **a** = **f** which is a more declarative and reusable form than if separate functions were needed for matrix multiplication and equation solving, respectively. It is also a more optimizable form since the optimizer can find ways to execute the statement which would be hidden if the user explicitly had stated in which direction the `*` function is executed.

Hence, multi-directional functions relieve the programmer from having to define all different variants of the operator in the query language and also from deciding which variant to use in a particular situation. The latter is the task of the query optimizer which through multi-directional functions is given more degrees of freedom for optimization.

```
DECLARE K AS SymmetricMx;
DECLARE f AS ColumnMx;
SELECT a FROM ColumnMx a WHERE K * a = f;
```

Fig. 7. A simple query illustrating function overloading.

The implementor can often define different implementations of `*` depending on if an argument or the result is unknown. For square matrices, □ × □ = □, two variants are required. The first variant does matrix multiplication when both arguments are known and the result is unknown. When the first argument and the result are known and the second argument is unknown, `*` will perform equation solving using Gauss decomposition. There are also two variants for symmetric matrices, ⧅ × □ = □, the difference is that instead of using Gauss decomposition when the second argument is unknown, the more efficient **LDL**^{T} decomposition is used.

Late binding relieves the programmer from having to decide when any of the more specialized multiplication operators can be used since the system will do this at run-time. Thus, the general matrix multiplication will at run-time be selected as the most specialized variant possible, e.g. ⧅ × □ = □, when the first argument is a symmetric matrix. Note that the types of all arguments participate in type resolution to select which resolvents of the multiplication operator to use from all the possible multiplication operators in Fig. 5. Contrast this with a pure object-oriented data model without multi-methods where the system cannot select the correct resolvent when the type of another argument than the first one has to be considered. This imposes restrictions on how object relations can be expressed in a pure object-oriented data model. In some models, e.g. C++, the types of all arguments are used to select

resolvent if the function is early bound but not when it is late bound. Thus, the introduction of an overriding function may become problematic.

Our example shows that multi-directional functions are useful to support the modeling of complex applications. A system that supports both late binding and multi-directional multi-methods offers the programmer a flexible and powerful modeling tool. It is then the challenge to provide query processing techniques to support these features. This will be addressed next.

5 Processing Queries with Multi-Directional Functions

We will show through an example how queries with multi-directional functions are processed in AMOS for a subset of the functions in Fig. 5. In Fig. 8 the function definitions in AMOSQL are given that are needed for the previous example in Fig. 6.

```
CREATE FUNCTION factorise(SymmetricMx K) ->
                  <DiagonalMx D, UUnitMx U> AS
                 FOREIGN Factorise;
CREATE FUNCTION transpose(UUnitMx U) -> LUnitMx L AS
                 FOREIGN Transpose;
CREATE FUNCTION times(LUnitMx L, ColumnMx y) -> ColumnMx f AS
                 MULTIDIRECTIONAL
                   'bbf' FOREIGN LUnitMult
                   'bfb' FOREIGN LUnitSolve;
CREATE FUNCTION times(DiagonalMx D, ColumnMx x) -> ColumnMx y AS
                 MULTIDIRECTIONAL
                   'bbf' FOREIGN DiagonalMult
                   'bfb' FOREIGN DiagonalSolve;
CREATE FUNCTION times(UUnitMx U, ColumnMx a) -> ColumnMx x AS
                 MULTIDIRECTIONAL
                   'bbf' FOREIGN UUnitMult
                   'bfb' FOREIGN UUnitSolve;
CREATE FUNCTION times(SymmetricMx K, ColumnMx a) -> ColumnMx f AS
                 MULTIDIRECTIONAL
                 'bbf' FOREIGN SymmetricMult,
                 'bfb' AMOSQL SymmetricSolve;
CREATE FUNCTION SymmetricSolve(SymmetricMx K,ColumnMx f)->ColumnMx
                 AS SELECT a
                 FROM UUnitMx U,DiagonalMx D,ColumnMx x,ColumnMx y
                   WHERE
                     factorise(K) = <U,D> AND
                     transpose(U) * y = f AND
                     D * x = y AND
                     U * a = x;
```

Fig. 8. Multi-method function definitions needed in the example

The multi-directional function `times` is overloaded and defined differently depending on the shape and representation of the matrixes. It is multi-directional to handle both matrix multiplication and equation solving. The primitive matrix operations are implemented as a set of User Defined Functions (UDFs) in some programming language (i.e. C). For symmetric matrixes the multiplication is implemented by the UDF `SymmetricMult` while equation solving uses the LDL method (`SymmetricSolve`) above.

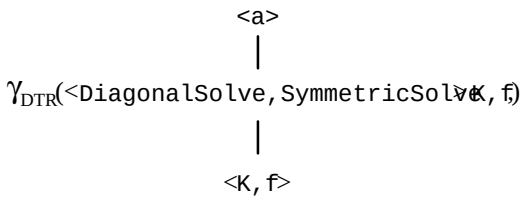

Fig. 9. The top level query algebra tree for the query in Fig. 7

The query optimizer translates the query into an optimized execution plan represented as an algebra tree (Fig. 9). During the translation to the algebra the optimizer will apply type resolution methods to avoid late binding in the execution plan when possible. In cases where late binding is required the execution plan may call subplans.

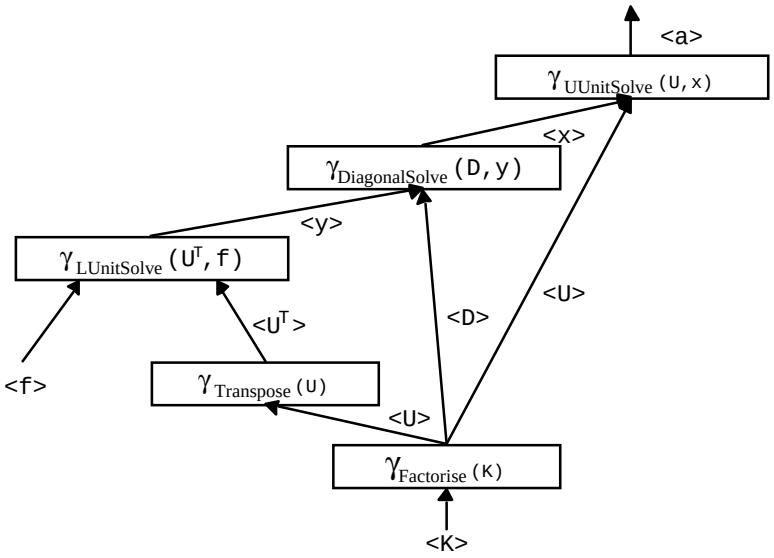

Fig. 10. Algebra tree showing the execution order for the transformed matrix expression

In the example query of Fig. 7 there are two possible resolvents of the name `times`:

1. `DiagonalMx.ColumnMx.times→ColumnMx`
2. `SymmetricMx.ColumnMx.times→ColumnMx`

The example thus requires late binding where resolvent 2 will be chosen when the first argument is a `SymmetricMx` and 1 will be chosen otherwise. When resolvent 2 is chosen the system will be solved using the **LDL**^{T}-decomposition, while a trivial diagonal solution method is used for case 1. The optimizer translates the query into the algebra tree in Fig. 9, where the algebra operator γ_{DTR} implements late binding. It selects at run time the subplan to apply on its arguments `K` and `f` based on their types. In the example there are two subplans, `DiagonalSolve` and `SymmetricSolve`. `DiagonalSolve` is implemented as a foreign function in C (Fig. 8), while `SymmetricSolve` references the subplan performing the **LDL**^{T}-decomposition in Fig. 10.

The subplan in Fig. 10 has `K` and `f` as input parameters (tuples) and produces `a` as the result tuple. `K` is input to the application of the foreign function `factorise` that does the **LDL**^{T} factorization producing the output tuple `<D,U>`. The `U` part is projected as the argument of the functions `transpose` and `UUnitSolve` (the projection operator is omitted for clarity), while the projection of `D` is argument to `DiagonalSolve`. The application of `LUnitSolve` has the arguments `f` and the result

of `transpose(U)`, i.e. U^T. Analogous applications are made for `DiagonalSolve` and `UUnitSolve` to produce the result tuple `<a>`.

6 Summary and Future Work

Domain-oriented data representations are needed when representing and querying data for numerical application using an ORDBMS, e.g. to store matrices. To avoid unnecessary data transformations and transmissions, it is important that the query language can be extended with domain-oriented operators, e.g. for matrix calculus. For high-level application modeling and querying of, e.g., matrix operators multi-directional functions are required.

Although multi-directional functions require generalized type checking, the benefits gained as increased naturalness and modeling power of the data model are important for many applications, including ours.

Our example illustrates how multi-directional functions can be utilized by the query optimizer. This is a new area for query optimization techniques.

We also showed that the system must support late binding for multi-directional functions. In our approach each late bound function in a query is substituted with a `DTR` calculus operator which is defined in terms of the resolvents eligible for execution. Each `DTR` call is then translated to the algebra operator γ_{DTR} that chooses among eligible subplans according to the types of its arguments. Local query execution plans are generated at each application of γ_{DTR} and optimized for the specific binding-patterns that will be used at run-time, as illustrated in our examples.

Further interesting optimization techniques to be investigated include the identification of common subexpressions among the possible resolvents, transformations of `DTR` expressions, and performance evaluation.

References

1 Amiel, E., Dujardin, E.: Supporting Explicit Disambiguation of Multi-Methods. Research Report n2590, Inria, 1995. ECOOP96, 1996.

2 Agrawal, R., DeMichiel, L.G., Lindsay, B.G.: Static Type Checking of Multi-Methods. OOPSLA Conf., p113-128, 1991.

3 Atkinson, M., Bancilhon,F., DeWitt,D., Dittrich,K., Maier,D., Zdonik,S.: The Object-Oriented Database System Manifesto. 1st. Conf. Deductive OO Databases, Kyoto, Jpn, Dec. 1989.

4 Banerjee, J., Kim, W., Kim, K.C.: Queries in Object-Oriented Databases. IEEE Data Eng. Conf., Feb. 1988.

5 Bertino, E., Negri, M., Pelagatti, Sbattella, G., L.: Object-Oriented Query Languages: The Notion and the Issues. IEEE Trans. on Knowledge and Data Eng., v4, Jun 1992.

6 Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., Zdybel, F.: CommonLoops Merging Lisp and Object-Oriented Programming. OOPSLA Conf., 1986.

7 Cardelli, L., Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys, v17, 1985.

8 Cattell, R. G. G.: Object Data Management: Object-Oriented and Extended Relational Database Systems. Addison-Wesley, 1992.
9 Cattell,R.G.G. (ed.): The Object Database Standard: ODMG 2.0. Morgan Kaufmann 1997.
10 Chambers, C., Leavens, G. T.: Typechecking and Modules for Multi-Methods. OOPSLA Conf., Oct 1994.
11 Chen, W., Turau, V., Klas, W.: Efficient Dynamic Look-Up Strategy for Multi-Methods. 8th European Conf., ECOOP94, Bologna, Italy, Jul 1994, Lecture Notes in Computer Science, N821, p408-431, Springer Verlag, 1994.
12 Daniels, S., Graefe, G., Keller,T., Maier,D., Schmidt, D., Vance, B.: Query Optimization in Revelation, an Overview. IEEE Data Eng. Bulletin, v14, p58-62, Jun 1992.
13 Fahl, G., Risch, T., Sköld, M.: AMOS, an Architecture for Active Mediators. Int. Workshop on Next Generation Information Techn. and Systems, Haifa, Israel, Jun 1993.
14 Flodin, S.: An Incremental Query Compiler with Resolution of Late Binding. LITH-IDA-R-94-46, Dept of Computer and Information Science, Linköping Univ., 1994.
15 Flodin, S.: Efficient Management of Object-Oriented Queries with Late Bound Functions. Lic. Thesis 538, Dept. of Computer and Inf. Science, Linköping Univ., Feb. 1996.
16 Flodin, S., Risch, T.: Processing Object-Oriented Queries with Invertible Late Bound Functions. VLDB 1995, Sept. 1995.
17 Flodin, S., Karlsson, J., Risch, T., Sköld, M., Werner, M.: AMOS User's Guide. CAELAB Memo 94-01, Linköping Univ., 1994.
18 Golub, G.H., van Loan, C.F.: Matrix Computations 2ed, John Hopkins Univ. Pr., 1989.
19 Graefe, G.: Query Evaluation Techniques for Large Databases. ACM Computing Surveys, v25, pp 73-170, June 1993.
20 IBM: DB2 Universal Database SQL Reference Ver. 5. Document S10J-8165-00. 1997.
21 Libkin,L., Machlin,R., Wong,L.: A Query Language for Multidimensional Arrays: Design, Implementation and Optimization Techniques. ACM SIGMOD, p228-239, June 1996.
22 Litwin, W., Risch, T.: Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates. IEEE Trans. on Knowledge and Data Eng., v4, Dec. 1992.
23 Lyngbaek, P.: OSQL: A Language for Object Databases. HPL-DTD-91-4, HP, Jan 1991.
24 Melton, J. (ed), ANSI SQL3 Papers SC21N9463-SC21N9467, ANSI SC21, NY, USA 1995.
25 Orsborn, K.: Applying Next Generation Object-Oriented DBMS to Finite Element Analysis. 1st Int. Conf. on Applications of Databases, ADB94, Lecture Notes in Computer Science, 819, p215-233, Springer Verlag, 1994.
26 Orsborn, K., Risch, T.: Next Generation of O-O Database Techniques in Finite Element Analysis. 3rd Int. Conf. on Computational Structures Technology (CST96), Budapest, Hungary, August, 1996.
27 Orsborn, K.: On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications. PhD Thesis 452, ISBN9178718279, Linköping Univ., Oct. 1996.
28 Ross, T. J., Wagner, L. R., Luger, G. F.: Object-Oriented Programming for Scientific Codes. II: Examples in C++. Journal of Computing in Civil Eng., v6, p497-514, 1992.
29 Scholz, S. P.: Elements of an Object-Oriented FEM++ Program in C++. Computers & Structures, v43, p517-529, 1992.
30 Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access Path Selection in a Relational Database Management System. ACM SIGMOD p23-34 1979.
31 Shipman, D. W.: The Functional Data Model and the Data Language DAPLEX. ACM Transactions on Database Systems, v6, March 1981.
32 Stonebraker, M.: Implementation of Integrity Constraints and Views by Query Modification. ACM SIGMOD , San Jose, CA, May 1975.
33 Stonebraker, M.: Object-relational DBMSs, Morgan Kaufmann, 1996.

34 Straube, D. D., Özsu, M. T.: Queries and Query Processing in Object-Oriented Database Systems. ACM Transactions on Information Systems, v8, Oct. 1990.
35 Vandenberg, S., DeWitt, D.: Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. ACM SIGMOD, 1991.