Representing Matrices Using Multi-Directional Foreign Functions *

Kjell Orsborn
Uppsala University
Kjell.Orsborn@it.uu.se

Tore Risch
Uppsala University
Tore.Risch@it.uu.se

Staffan Flodin
OzEmail Ltd, Australia
Staffan.Flodin@team.ozemail.com.au

2004

Abstract

New application areas for database technology such as computer-aided design and analysis systems require a semantically rich data model, high performance, extensibility, and application oriented queries. We have built a system for finite element analysis using a functional database management system to represent multiple matrix representations. The functional data model is well suited for managing the data complexity and representation needs of such applications. Our application domain needs multiple customized numerical data representations, user-defined (foreign) functions, multiply inherited types, and overloading of multi-argument functions. Type inheritance and overloading requires the support for late binding of function calls in queries. Furthermore, queries may use functions as relationships which means that the system needs to process inverses of both tabulated and foreign functions, *multi-directional functions*. It is shown how to model matrix algebra operators using multi-directional foreign functions and how to process queries to these functions.

1 Introduction

As the usage of database techniques will increase in scientific and engineering disciplines the requirements on analysis capabilities will grow. This work provides analysis capabilities in a database environment to support the needs of such applications. Scientific and engineering applications are computationally

*Published in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): *Functional Approach to Computing with Data*, Springer, ISBN 3-540-00375-4, 2004.

intensive applications and the idea is to provide numerical analysis capabilities within the database environment to support their processing needs.

By providing in the DBMS both application oriented data structures and the corresponding implementations of relevant operations, it is possible to both store and process the data in the DBMS. In this way data transportation between application and database can be minimized. Furthermore, the embedding of local databases within applications can provide powerful new techniques for developing advanced applications.

For example, we would like to store numerical matrices in the database, not only the array data structure. This makes it possible to perform operations on matrices producing new or modified matrices, in contrast to only accessing the physical array data structure. By having matrix types in the database it is possible to extend the query language with operations on matrices to form an algebra for the matrix domain that can be used in application modeling. It is furthermore possible to introduce special query optimization methods for numerical data processing. This includes, e.g., decisions for selecting suitable data representations, solution methods, and processing locations.

We have built a system [19, 20, 21] for finite-element analysis (FEA) storing numerical data in the functional DBMS Amos II [22]. The performance requirements on numerical data access in an FEA model are very high, and several special methods have been developed for efficient representation of, e.g., matrices used in FEA where different methods are useful depending on the context. For instance, a fundamental component in an FEA system is an equation solving sub-system that can solve linear equation systems such as $K \times a = f$ where a is sought while K and f are known. Special representation methods and special equation solving methods must be applied dependent on the properties and representations of the matrices. To get good performance in queries involving matrix operators, various matrix data representations and function definitions are stored in the database and the functions are executed in the database server. Special optimization methods and cost formulae have been developed for the matrix domain operators.

Views in our functional data model are represented as functions defined in terms of queries containing functions from the domain, e.g. for matrix algebra. In our query language, AmosQL [22], such functions are called *derived functions* and are expressed by side-effect free queries. This provides a high abstraction level which is problem oriented and reusable. Query optimization techniques are used to optimize queries involving derived functions.

In simple object-oriented models the method invocation is based on the *message-passing* paradigm where methods are connected to the receiver of the message. 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 matrix domain it is desirable to also support functions with more than one argument, corresponding to *multi-methods* [2] in object-oriented languages. This provides a natural way to represent, e.g., matrix operators applied on various kinds of matrix representations. In Sect. 3 we show how matrix operators are modeled as multi-argument

```
DECLARE m1 AS SymmetricMatrix;
DECLARE m2 AS ColumnMatrix;

SELECT x FROM ColumnMatrix x WHERE x IN f() AND m1 * x = m2;
```

Figure 1: A sample query exemplifying a multi-directional method

overloaded functions in our application.

High level declarative queries with function calls do not specify exactly how a function is to be invoked in the query. We will show that *multi-directional foreign functions* [15] are needed in the database query language for efficient 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 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 query in Fig. 1 that retrieves those matrices stored in function `f()` which, when multiplied by the matrix bound to the variable `m1`, equals the matrix bound to the variable `m2`. This query is a declarative specification of the retrieval of the matrix `x` from the result set of the function h`f()` where `x` solves the equation system `m1 * x = m2` (`m1` and `m2` are assumed given). It is the task of the query optimizer to find an efficient execution strategy for the declarative specification, e.g. by using the inverse of the `*` method (matrix multiplication) that solves the equation system to get a value for `x`. The alternative execution strategy without multi-directional foreign functions is to go through all matrices in `f()` and multiply them with `m1` to compare the result with `m2`. The first strategy clearly scales substantially better when `f()` contains large sets of matrices.

To extend the search space of a query optimizer it must be able to inspect the definitions of all referenced views. In our case this means that the optimizer is revealed the definitions of derived functions to be able to in-line them before queries can be fully optimized, a process called *revelation* [7]. With revelation the query optimizer is allowed break encapsulation while the user still cannot access encapsulated data.

The combination of inheritance in the type hierarchy and function 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, function inverses, 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 special query processing techniques are not applied. This is why providing a solution that enables optimization of late bound functions is an important issue in the context of a database [7].

A special problem is the combination of late bound functions and multi-directional functions. This problem is addressed in [8] where late bound function calls are represented by a special object 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 (inverse) resolvents. The single argument `DTR` approach in [8] has been generalized to handle multi-directional functions with arbitrary arity [9].

Below is is shown, by using excerpts from our matrix domain, that using a functional 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 Functional database representations

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.

2.1 The functional data model

The functional data model of Amos II [22] and its query language AmosQL is based on DAPLEX [25] with object-oriented extensions. The data model includes *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 through queries 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 II also has *foreign functions* which are defined using an auxiliary programming language such as Java, Lisp, or C and then introduced into the query language [15]. In Fig. 1 the overloaded operator `*` over type `ColumnMatrix` 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.

2.2 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-*

```
CREATE FUNCTION times(SymmetricMatrix x, ColumnMatrix y) ->
                 ColumnMatrix r AS
                 MULTIDIRECTIONAL
                 "bbf" FOREIGN "MatrixMultiplication"
                       COST "MultCost",
                 "bfb" FOREIGN "GaussDecomposition"
                       COST "GaussCost";
```

Figure 2: Definition of a multi-directional foreign function

```
SELECT x FROM ColumnMatrix x WHERE m1 * x = m2;
```

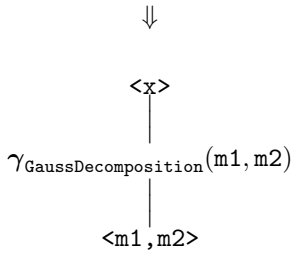

Figure 3: Multi-directional function execution

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_i^b or a_i^f if a_i is bound or free, respectively. In the optimal execution plan for the query 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 $x^b \times y^f \rightarrow r^b$.

Recall the three types of functions in the Amos II data model: stored, foreign and derived functions as decribed in a separate chapter on Amos II . 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 illustrates a very simple AmosQL query and its translation to object algebra. Notice that the terms following the `FROM` clause denote declarations of typed variables universally quantified over type extents [22]. This is different

from OQL [6] where they denote collections and SQL where they denote tables.

In Fig. 2 the function `times` (implementing `*`) is defined for two arguments and made multi-directional by defining which implementation to use for a certain binding-pattern. For $times(a^b, b^f) \rightarrow r^b$ (implementing $x^b \times y^f \rightarrow r^b$) 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 both selectivities and costs.

The query compiler 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 as shown in Fig. 2. The chosen implementation of `times`, i.e. `GaussDecomposition`, depends on the types of `m1` and `m2`. 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.

2.3 Function overloading and late binding

In Amos II's data model types are organized in a hierarchy with inheritance where subtypes inherit properties from their supertypes. *Overloading* allows the function name denote several variants, called *resolvents*. Resolvents are uniquely named by annotating the function name with the type of the arguments and result. The naming convention chosen in Amos II (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 functional model an instance of type t is also an instance of all supertypes of that type, i.e. *inclusion polymorphism* [4]. 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-argument functions the criteria for late binding is similar as for pure object-oriented methods with the difference that for multi-argument functions, *tuple types* are considered instead of single types. Multi-argument functions enhance the expressive power of the data model but the type checker must include algorithms for type resolution of tuple types [2] and for handling ambiguities [1].

In Amos II the query compiler resolves which function calls require late binding. Whenever late binding is required a special operator, `DTR` [8], is inserted into the calculus expression. The optimizer translates each `DTR` call into the special algebra operator $\gamma_{\texttt{DTR}}$ 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 [11] to find an efficient execution strategy.

3 Representing matrices using functions

It will be investigated how to represent matrices and their operators using our semantic functional data model, and how to process queries over these representations.

3.1 Matrix algebraic concepts

Some matrix algebraic concepts are introduced where the notation mainly follows that of Golub and van Loan [10]. The vector space of all m-by-n matrices is denoted by the m-by-n scalar field $S^{m\times n}$, where normally $S \in R$, the set of real numbers. However, due to the computational requirements it might be necessary to extend the types of matrix representations such that S belongs to one of Z (the set of integers), R_f (the set of four-byte reals), and R_d (the set of 8-byte reals). This means that matrices can have integer, float, and double representations. This must be taken care of in the definition of matrix operations for allowing matrix expressions mixing matrix types. This distinction is left out in the subsequent presentation of matrix concepts.

Thus, for a matrix $\mathbf{A}$ we have,

$$\mathbf{A} \in S^{m\times n} \Leftrightarrow \mathbf{A} = (a_{ij}) = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}, \text{ where } a_{ij} \in S\,. \qquad (1)$$

Here, a_{ij} represents the element of $\mathbf{A}$ at row i and column j.

Basic matrix algebraic operations of matrices can now be introduced. The conventional approach introduces matrix algebraic operations as functions that take matrices as arguments and produce new or altered matrices. Here, a somewhat different approach will be applied. Since the query language AmosQL allows the definition of multi-directional functions, it is possible to define operations on matrices as relationships that are isomorphic to the corresponding

mathematical expressions. In Golub and van Loan, [10], operations are represented by the $a \rightarrow b$ notation, where the arrow associates to a one-directional function application. In the present context, this notation is replaced by the $a \leftrightarrow b$ notation that is more associated to bi-directional or multi-directional relationships. However, it should be noted that this notation does not imply that the relationship exist, or is defined, for every direction that corresponds to combinations of matrix types.

Hence, the basic operations on matrices include:

- **addition**: $S^{m\times n} \times S^{m\times n} \leftrightarrow S^{m\times n}$, where $\mathbf{A} + \mathbf{B} = \mathbf{C}$ with the elements $a_{ij} + b_{ij} = c_{ij}$
- **subtraction**: $S^{m\times n} \times S^{m\times n} \leftrightarrow S^{m\times n}$, where $\mathbf{A} - \mathbf{B} = \mathbf{C}$ with the elements $a_{ij} - b_{ij} = c_{ij}$
- **multiplication**: $S^{m\times r} \times S^{r\times n} \leftrightarrow S^{m\times n}$, where $\mathbf{A} \times \mathbf{B} = \mathbf{C}$ with the elements $a_{ij} \cdot b_{ij} = c_{ij}$
- **transposition**: $S^{m\times n} \leftrightarrow S^{n\times m}$, where $\mathbf{A}^T = \mathbf{B}$ with the elements $a_{ij} = b_{ji}$.

We should note that the matrix concept defined above covers general m-by-n matrices. By making restrictions on this definition it is possible to define specialised categories of matrices that form subspaces of the vector space $S^{m\times n}$. For instance, we can define:

- $S^{m\times n}$, representing the general **rectangular** matrix, $\mathbf{A}_{rect}$.
- $S^{m\times m}$, representing a **square** matrix, $\mathbf{A}_{square}$, with the same number of rows and columns.
- $S^{m\times m}$, a square matrix with the additional constraint $s_{ij} = s_{ji}$ that represents a **symmetric** matrix, $\mathbf{A}_{symm}$.
- $S^{m\times m}$, a symmetric matrix with the additional constraint $s_{ij} = 0$ for $i \neq j$ that represents a **diagonal** matrix, $\mathbf{A}_{diag}$.
- $S^{m\times m}$, a matrix with the same number of rows and columns with the additional constraint $s_{ij} = 0$ for $i > j$ and that represents an **upper triangular** matrix, $\mathbf{A}_{uptri}$.
- $S^{m\times m}$, an upper triangular matrix with the additional constraint $s_{ij} = 1$ for $i = j$ that represents an **upper unit triangular** matrix, $\mathbf{A}_{uputri}$.
- $S^{m\times m}$, a matrix with the same number of rows and columns with the additional constraint $s_{ij} = 0$ for $i < j$ and that represents a **lower triangular** matrix, $\mathbf{A}_{lowtri}$.
- $S^{m\times m}$, an lower triangular matrix with the additional constraint $s_{ij} = 1$ for $i = j$ that represents a **lower unit triangular** matrix, $\mathbf{A}_{lowutri}$.

- $S^{m\times 1}$, a rectangular matrix with 1 column representing a **column** matrix, $\mathbf{a}$ or $\mathbf{A}_{col}$.
- $S^{1\times m}$, a rectangular matrix with 1 row representing a **row** matrix type, $\mathbf{a}$ or $\mathbf{A}_{row}$.

With these additional categories of matrices, the previous list of matrix operations can also be specialised further taking the additional categories into account. This is exemplified in Eqs. 2-19 for the case of matrix multiplication of rectangular matrices where index sizes and symmetries have been used to identify different combinations.

$$\begin{aligned}
\mathbf{A}_{rect} \times \mathbf{B}_{rect} &= \mathbf{C}_{rect} \ , && (2)\\
\mathbf{A}_{rect} \times \mathbf{B}_{square} &= \mathbf{C}_{rect} \ , && (3)\\
\mathbf{A}_{square} \times \mathbf{B}_{rect} &= \mathbf{C}_{rect} \ , && (4)\\
\mathbf{A}_{symm} \times \mathbf{B}_{rect} &= \mathbf{C}_{rect} \ , && (5)\\
\mathbf{A}_{square} \times \mathbf{B}_{col} &= \mathbf{C}_{col} \ , && (6)\\
\mathbf{A}_{symm} \times \mathbf{B}_{col} &= \mathbf{C}_{col} \ , && (7)\\
\mathbf{A}_{lowtri} \times \mathbf{B}_{col} &= \mathbf{C}_{col} \ , && (8)\\
\mathbf{A}_{uptri} \times \mathbf{B}_{col} &= \mathbf{C}_{col} \ , && (9)\\
\mathbf{A}_{lowtri} \times \mathbf{B}_{uptri} &= \mathbf{C}_{square} \ , && (10)\\
\mathbf{A}_{diag} \times \mathbf{B}_{col} &= \mathbf{C}_{col} \ , && (11)\\
\mathbf{A}_{diag} \times \mathbf{B}_{uptri} &= \mathbf{C}_{uptri} \ , && (12)\\
\mathbf{A}_{col} \times \mathbf{B}_{row} &= \mathbf{C}_{rect} \ , && (13)\\
\mathbf{A}_{rect} \times \mathbf{B}_{rect} &= \mathbf{C}_{square} \ , && (14)\\
\mathbf{A}_{square} \times \mathbf{B}_{square} &= \mathbf{C}_{square} \ , && (15)\\
\mathbf{A}_{col} \times \mathbf{B}_{row} &= \mathbf{C}_{square} \ , && (16)\\
\mathbf{A}_{rect} \times \mathbf{B}_{col} &= \mathbf{C}_{col} \ , && (17)\\
\mathbf{A}_{row} \times \mathbf{B}_{rect} &= \mathbf{C}_{row} \ , && (18)\\
\mathbf{A}_{row} \times \mathbf{B}_{square} &= \mathbf{C}_{row} \ . && (19)
\end{aligned}$$

Hence, the resulting matrix category of multiplying two (rectangular) matrices is dependent on the sizes of the outer indexes of the argument matrices. By interpreting the matrix spaces as sub-categories of the rectangular matrix category we get relationships between argument matrix categories and result argument category for the matrix multiplication operator. By considering other matrix characteristics, such as symmetry and singularity, further specialisations of these relationships can be established.

Furthermore, in applications like FEA, it is common to use specialised and more compact physical representations [13, 5] of matrices in contrast to *full* regular matrix representations. These type of compressed representations include,

for example, skyline matrix (or profile matrix) representations where consecutive zero-valued elements above the skyline are left out and the matrix is usually represented by matrix columns in a one-dimensional array. This is an example of a compact representation where the matrix structure is static, i.e. it is not allowed to change. Additional static representations along the same theme exists. There are also dynamic matrix representations where the storage structure is allowed to change. These representation types are usually referred to as sparse matrix representations and are typically implemented by some linked-list data structure. The categorisations and their usage in establishing the multiplication operator as relationships among different categories that are exemplified above can be further extended to establish relationships between combinations of other matrix categories and representations as well as for different operators.

To sum up this part, three principles have been presented that can divide the matrix concept into different categories, namely:

- mathematically-related matrix categories based on the general matrix concept and its characteristics that further restrict this concept into subcategories.
- the data types, integer, float, and double, used for representing and implementing numerical matrices.
- various physical representations schemes for representing and implementing matrices such as regular, skyline, or sparse. A database implementation that covers regular and skyline representations is presented in [21].

The reason for defining several matrix categories is the potential ability to take advantage of the knowledge about specific categories in representing numerical data and applying numerical analysis methods. This concerns the possibilities of applying efficient storage and processing techniques. So far it is possible to use:

- a priori information to determine matrix categories appropriate for a specific problem.
- information about matrix categories related by a specific operator to determine appropriate operator and the correct result (or argument) category.
- information about matrix characteristics, i.e. properties that are not distinguished by separate categories, to determine correct operator result (or argument) to efficiently direct subsequent matrix representations and operations.

This type of information has been used to establish the matrix type structure in Fig. 4 together with the set of mathematical operations that will be discussed below.

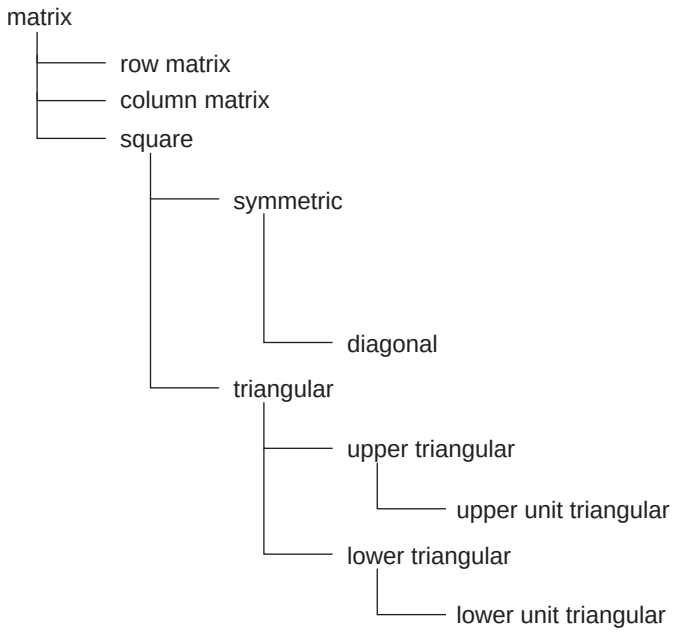

Figure 4: A matrix type hierarchy for linear matrix algebra

3.2 Queries for solving linear equations

A fundamental component in an FEA system is a linear equation solving subsystem. To model an equation solving system we define a type hierarchy of matrix types as illustrated in Fig. 4. Furthermore, the matrix multiplication operator, $\times$, is defined as functions on this matrix type hierarchy for several combinations of arguments. Eqs. 2-19 illustrate how each variant of the multiplication function takes various matrix types as arguments.

The functions for multiplication are used to specify linear equation systems as matrix multiplications as $\mathbf{K} \times \mathbf{a} = \mathbf{f}$, where $\mathbf{a}$ is sought while $\mathbf{K}$ and $\mathbf{f}$ 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 $\mathbf{K}$ is a symmetric matrix, i.e. corresponding to the case in Eq. 7, it can be solved by a method that explores the symmetric properties of the first argument. One such method, $\mathbf{LDL}^{\mathrm{T}}$ decomposition [10] outlined in Eq. 20, substitutes the equation system with several equivalent equation systems that are simpler to solve. According to Eq. 20 the left-hand equation can be transformed into a set of simpler equations on the right-hand side.

```
DECLARE K AS SymmetricMatrix;
DECLARE f AS ColumnMatrix;

SELECT a FROM ColumnMatrix a WHERE K * a = f;
```

Figure 5: A simple query illustrating function overloading

$$\mathbf{K}^b \times \mathbf{a}^f = \mathbf{f}^b \longrightarrow \begin{cases} \mathbf{K}^b & = & (\mathbf{U}^T)^f \times \mathbf{D}^f \times \mathbf{U}^f \\ \mathbf{U}^b \times \mathbf{a}^f & = & \mathbf{x}^b \\ \mathbf{D}^b \times \mathbf{x}^f & = & \mathbf{y}^b \\ (\mathbf{U}^T)^b \times \mathbf{y}^f & = & \mathbf{f}^b \end{cases} . \quad (20)$$

The linear equation system is solved by starting with the factorization, $\mathbf{K} = \mathbf{U}^T \times \mathbf{D} \times \mathbf{U}$, that transforms $\mathbf{K}$ into the three matrices $\mathbf{U}^T$, $\mathbf{D}$, and $\mathbf{U}$. Then the upper triangular equation system $\mathbf{U}^T \times \mathbf{y} = \mathbf{f}$ is solved to get $\mathbf{y}$. The diagonal equation system $\mathbf{D} \times \mathbf{x} = \mathbf{y}$ is then solved to get $\mathbf{x}$, and finally the solution of the lower triangular equation system $\mathbf{U} \times \mathbf{a} = \mathbf{x}$ gets $\mathbf{a}$. If the equation on the other hand corresponds to Eq. 6, the symmetry of $\mathbf{K}$ cannot be exploited and some other method to solve it must be applied, e.g. Gauss decomposition.

The rationale for having these different overloaded matrix multiplication functions is efficiency and reusability. Efficiency because mathematical properties of the more specialised matrix types can be considered when implementing multiplication operators for them. Furthermore, specialised 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 specialised 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. 5 over the type hierarchy in Fig. 4 and the multiplication operators from Eqs. 2-19.

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 `SquareMatrix` will be selected at run-time, i.e. late binding. Also note that here the multiplication function (`*`) is overloaded and 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. 5 can be written as $\mathbf{K} \times \mathbf{a} = \mathbf{f}$ 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.

The implementor can often define different implementations of the `times` (`*`) operation depending on whether an argument or the result is unknown. For the case in Eq. 6, where a square matrix is multiplied with a column matrix resulting in another column matrix, 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, the `times` (`*`) operation will perform equation solving using Gauss decomposition. There are also two variants for symmetric matrices, Eq. 7, the difference is that instead of using Gauss decomposition when the second argument is unknown, the more efficient $\mathbf{LDL}^{\mathrm{T}}$ decomposition algorithm is used.

Late binding relieves the user from having to decide when any of the more specialised 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 specialised variant possible, e.g. a variant corresponding to Eq. 7 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 Eqs. 2-19. Contrast this with a pure object-oriented data model without multi-argument functions 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-argument functions 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.

4 Processing queries with multi-directional functions

We will show through an example how queries with multi-directional functions are processed in Amos II for a subset of the functions corresponding to Eqs. 2-19. In Fig. 6 the function definitions in AmosQL are given that are needed for modeling equation solving using the method of $\mathbf{LDL}^{\mathrm{T}}$ decomposition according to Eq. 20.

The multi-directional foreign function `times` is overloaded and defined differently depending on the shape and representation of the matrices. It is multi-directional to transparently handle both matrix multiplication and equation solving in queries. The primitive matrix operations are implemented as a set of foreign functions in some programming language (here C). For symmetric ma-

```
CREATE FUNCTION factorise(SymmetricMatrix K) ->
               <DiagonalMatrix D, UpUTriMatrix U> AS
               FOREIGN "Factorise";

CREATE FUNCTION transpose(UpUTriMatrix U) -> LowUTriMatrix L AS
               FOREIGN "Transpose";

CREATE FUNCTION times(LowUTriMatrix L, ColumnMatrix y) ->
               ColumnMatrix f AS
               MULTIDIRECTIONAL "bbf" FOREIGN "LowUTriMult",
                                "bfb" FOREIGN "LowUTriSolve";

CREATE FUNCTION times(DiagonalMatrix D, ColumnMatrix x) ->
               ColumnMatrix y AS
               MULTIDIRECTIONAL "bbf" FOREIGN "DiagonalMult",
                                "bfb" FOREIGN "DiagonalSolve";

CREATE FUNCTION times(UpUTriMatrix U, ColumnMatrix a) ->
               ColumnMatrix x AS
               MULTIDIRECTIONAL "bbf" FOREIGN "UpUTriMult",
                                "bfb" FOREIGN "UpUTriSolve";

CREATE FUNCTION times(SymmetricMatrix K, ColumnMatrix a) ->
               ColumnMatrix f AS
               MULTIDIRECTIONAL "bbf" FOREIGN "SymmetricMult",
                                "bfb" DERIVED "SymmetricSolve";

CREATE FUNCTION SymmetricSolve(SymmetricMatrix K,ColumnMatrix f) ->
               ColumnMatrix AS SELECT a
               FROM UpUTriMatrix U,DiagonalMatrix D,
                    ColumnMatrix x,ColumnMatrix y
               WHERE factorise(K) = <U,D> AND
                     transpose(U) * y = f AND
                     D * x = y AND
                     U * a = x;
```

Figure 6: Multi-directional function definitions

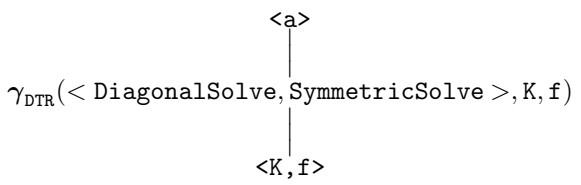

Figure 7: The top level query algebra tree for the query in Fig. 5

trices the multiplication is implemented by the foreign function `SymmetricMult` while equation solving uses the **LDL** method above, as specified by the derived function named (`SymmetricSolve`).

The query optimizer translates the query into an optimized execution plan represented as an algebra tree in Fig. 7. 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.

In the example query of Fig. 5 there are two possible resolvents of the name `times`:

1. `DiagonalMatrix.ColumnMatrix.times`→`ColumnMatrix`
2. `SymmetricMatrix.ColumnMatrix.times`→`ColumnMatrix`

The example thus requires late binding where resolvent (2) will be chosen when the first argument is a `SymmetricMatrix` and (1) will be chosen otherwise. When resolvent (2) is chosen the system will be solved using the $\mathbf{LDL}^{\mathrm{T}}$ decomposition, while a trivial diagonal solution method is used for case (1). The optimizer translates the query into the algebra tree in Fig. 7, where the algebra operator $\gamma_{\mathtt{DTR}}$ 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 shown in Fig. 6, while `SymmetricSolve` references the subplan performing the $\mathbf{LDL}^{\mathrm{T}}$ decomposition in Fig. 8.

The subplan in Fig. 8 has `K` and `f` as input parameters (tuples) and produces `a` as the result tuple. The `K` parameter is input to the application of the foreign function `Factorise` that does the $\mathbf{LDL}^{\mathrm{T}}$ factorization producing the output tuple `<D,U>`. The `U` part is projected as the argument of the functions `Transpose` and `UpUTriSolve` (the projection operator is omitted for clarity), while the projection of `D` is argument to `DiagonalSolve`. The application of `LowUTriSolve` has the arguments `f` and the result of `Transpose(U)`, i.e. $\mathtt{U}^{\mathrm{T}}$. Analogous applications are made for `DiagonalSolve` and `UpUTriSolve` to produce the result tuple `<a>`.

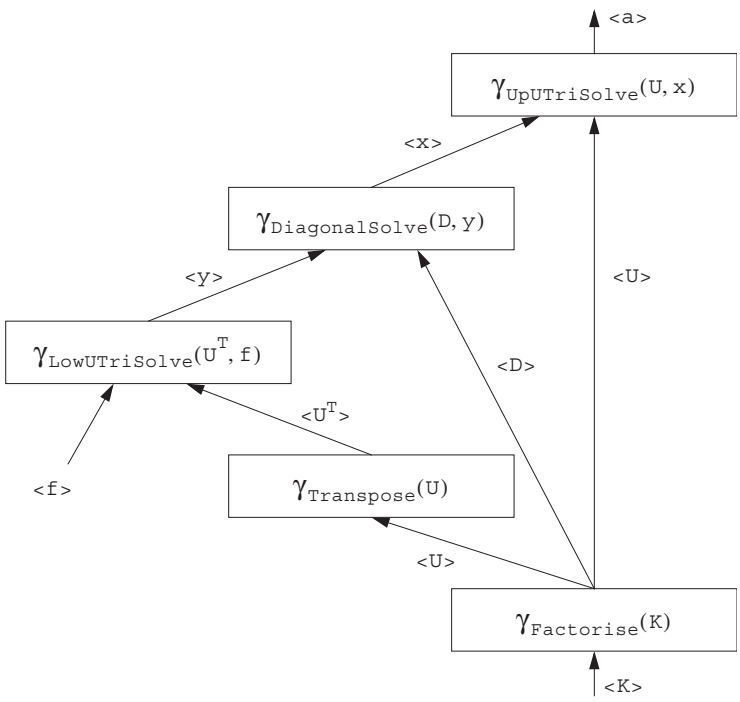

Figure 8: Algebra graph showing the execution order for the transformed matrix expression

5 Related work

Object-oriented database management systems (OODBMSs) are often motivated by advanced applications such as systems for management of scientific and engineering data [27, 3, 11]. In our case, a functional data model is shown to be even better suited for managing the complexity of the data in such applications.

Modeling matrix computations using the object-oriented paradigm has been addressed in e.g. [23, 24]. In [14, 18] algebras for primitive matrix operations are proposed. None of those papers address late binding, multi-methods, or multi-directional functions. We have shown 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, 27]. Several object-oriented query languages [6, 17, 16] have been proposed. However, queries with multi-directional functions have not been used in OODBMSs.

Multi-argument functions require generalized type resolution methods compared to the type resolution of pure object-oriented methods [1, 2]. In the database context the issue is not mainly fast type resolution when looking up methods but rather optimizing queries with expanded function definitions (views) in order to detect hidden search paths and multi-directional function inverses [8].

In the area of constraint programming [12] the user specifies *constraints* and then a constraint solver works out the best way to interpret them. Multi-directional foreign functions are a form of constraints compiled using a cost-based query optimizer for scalability over large data sets (large sets of matrices in the case presented here)

Advanced applications, such as our FEA application, require domain dependent data representations of matrices and an extensible object-relational query optimizer [26] to efficiently process queries over these representations. For accessing domain specific physical representations we have extended the technique of multi-directional foreign functions described in [15].

In [8] 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. Here that approach is generalized to multi-argument foreign functions [9].

6 Summary and future work

Domain-oriented data representations are needed when representing and querying data for numerical applications using a database system, 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-argument functions require generalized type checking and

query processing, the benefits gained as increased naturalness and modeling power of the data model are important for many applications, including ours.

It was shown how multi-directional and overloaded functions can be utilized by the query optimizer to scale the execution of queries over large sets of matrices. This is a new area for query optimization techniques.

The system must support optimization of queries with late binding for multi-directional functions in order for the proposed query optimization methods to be applicable. 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 $\gamma_{\texttt{DTR}}$ that chooses among eligible subplans according to the types of its arguments. Local query execution plans are generated at each application of $\gamma_{\texttt{DTR}}$ 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, query rewrite techniques for multi-directional functions, and transformations of `DTR` expressions.

References

[1] E.Amiel, E.Dujardin: Supporting Explicit Disambiguation of Multi-Methods. *ECOOP Conf.*, 1996.

[2] R.Agrawal, L.G.DeMichiel, B.G.Lindsay: Static Type Checking of Multi-Methods. *OOPSLA Conf.*, 113-128, 1991.

[3] E.Bertino, M.Negri, G.Pelagatti, L.Sbattella: Object-Oriented Query Languages: The Notion and the Issues. *IEEE Trans. on Knowledge and Data Engineering*, 4(3), 223-237, June 1992.

[4] L.Cardelli, P.Wegner: On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4), 471-522, 1985.

[5] G.F.Carey, J.T.Oden: *Finite Elements: Computational Aspects*, Prentice-Hall, Inc., Vol. 3, Texas Finite Element Series, 1984.

[6] R.G.G.Cattell (ed.): *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[7] S.Daniels, G.Graefe, T.Keller, D.Maier, D.Schmidt, B.Vance: Query Optimization in Revelation, an Overview. *IEEE Data Eng. Bulletin*, 14(2), 58-62, June 1992.

[8] S.Flodin, T.Risch: Processing Object-Oriented Queries with Invertible Late Bound Functions. *21st Conf. on Very Large Databases (VLDB'95)*, 335-344, 1995

[9] S.Flodin: *Efficient Management of Object-Oriented Queries with Late Bound Functions.* Licentiate Thesis 538, Dept. of Computer and Inf. Science, Linköping Univ., Feb. 1996.

[10] G.H.Golub, C.F.van Loan: *Matrix Computations 3rd ed.*, John Hopkins Univ. Pr., 1996.

[11] G.Graefe: Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2), 73-170, June 1993.

[12] P. van Hentenryck: Constraint Programming for Combinatorial Search Problems. *ACM Computing Surveys*, 28(4), 1996.

[13] J.T.H.Hughes: *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Prentice Hall International Ltd., 1987.

[14] L.Libkin, R.Machlin, L.Wong: A Query Language for Multidimensional Arrays: Design, Implementation and Optimization Techniques. *ACM SIGMOD conf.*, 228-239, June 1996.

[15] W.Litwin, T.Risch: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates. *IEEE Transactions on Knowledge and Data Engineering*, 4(6), 517-528, 1992

[16] P. Lyngbaek: *OSQL: A Language for Object Databases*, Tech. Report, HP Labs, HPL-DTD-91-4, 1991.

[17] A.Eisenberg, J.Melton: SQL:1999, formerly known as SQL 3, *SIGMOD Record*, 28(1), 131-138, 1999.

[18] A.P.Marathe, K.Salem: A Language for Manipulating Arrays, *VLDB Conf.*, 323-334, 1997.

[19] K.Orsborn: Applying Next Generation Object-Oriented DBMS to Finite Element Analysis. In W. Litwin, T. Risch (eds.): *Intl. Conf. on Applications of Databases (ADB'94)*, Springer, 215-233, 1994.

[20] K.Orsborn, T.Risch: Next Generation of O-O Database Techniques in Finite Element Analysis. *3rd Int. Conf. on Computational Structures Technology (CST96)*, Budapest, Hungary, August, 1996.

[21] K.Orsborn: *On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications.* PhD Thesis 452, ISBN9178718279, Linköping Univ., Oct. 1996.

[22] T.Risch, V.Josifovski, T.Katchaounov: Functional Data Integration in a Distributed Mediator System, in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): *Functional Approach to Computing with Data*, Springer, http://user.it.uu.se/~torer/publ/FuncMedPaper.pdf, 2004.

[23] T.J.Ross, L.R.Wagner, G.F.Luger: Object-Oriented Programming for Scientific Codes. II: Examples in C++. *Journal of Computing in Civil Eng.,*, Vol. 6, 497-514, 1992.

[24] S.P.Scholz: Elements of an Object-Oriented FEM++ Program in C++. *Computers & Structures,* Vol. 43, 517-529, 1992.

[25] D.Shipman: The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1), 140-173, 1981.

[26] M.Stonebraker, P.Brown: *Object-Relational DBMSs: Tracking the Next Great Wave.* Morgan Kaufmann Publishers, Inc., 1999.

[27] D.D.Straube, M.T.Özsu: Query Optimization and Execution Plan Generation in Object-Oriented Data Management Systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2), 210-227, 1995.