Chapter 10 Case Study: Generative Matrix Computation Library (GMCL) [155]

10.1 Domain Analysis

10.1.1 Domain Definition

10.1.1.1 Goals and Stakeholders

Our goal is to develop a *matrix computation library*. Thus, our domain is the *domain of matrix computation libraries*. The most important group of stakeholders are users solving linear algebra problems. We want the library to be highly reusable, adaptable, and very efficient (in terms of execution speed and memory consumption) and provide a highly intentional interface to application programmers. For our case study, we will always prefer technologically better solutions and ignore organizational issues. In a real world setting of a software developing organization, the analysis of stakeholders and their goals, strategic project goals, and other organizational issues may involve a significant effort.

10.1.1.2 Domain Scoping and Context Analysis

10.1.1.2.1 Characterization of the Domain of Matrix Computation Libraries

Our general domain of interest is referred to as *matrix computations*, which is a synonym for applied, algorithmic linear algebra. Matrix computations is a mature domain with a history of more than 30 years (e.g. [Wil61]). The domain includes both the well-defined mathematical theory of linear algebra as well as the knowledge about efficient implementations of algorithms and data structures for solving linear algebra problems on existing computer architectures. This implementation knowledge is well documented in the literature, e.g. [GL96, JK93].

In particular, we are interested in the *domain of matrix computation libraries*. A matrix computation library contains ADTs and algorithm families for matrix computations and is intended to be used as a part of a larger application. Thus, it is an example of a *horizontal* domain. Examples of *vertical* domains involving matrix computations would be matrix computation environments (e.g. Matlab [Pra95]) or specialized scientific workbenches (e.g. for electromagnetics or quantum chemistry). They are vertical domains since they contain entire applications including GUIs, graphical visualization, persistent storage for matrices, etc.

The main concepts in matrix computations are vectors, matrices, and computational methods, e.g. methods for solving a system of linear equations or computing the eigenvalues. A glossary of some of the terms used in matrix computations is given in 10.4.

10.1.1.2.2 Sources of Domain Knowledge

The following sources of domain knowledge were used in the analysis of the matrix computation libraries domain:

- literature on matrix computations: [GL96, JK93];
- documentation, source code, and articles describing the design of existing matrix computation libraries: [LHKK79, DDHH88, DDDH90, CHL+96] and those listed in Table 15 and Table 16;
- online repository of matrices: [MM].

10.1.1.2.3 Application Areas of Matrix Computation Libraries

In this section, we will identify features characterizing matrix computation libraries by analyzing different application areas of matrix computations.

Table 13 and Table 14 list some typical application areas of matrix computations and the types of matrices and computations which are required for solving the problems in the listed areas. The application areas were grouped into two categories: one requiring dense matrices (Table 13) and the other one requiring sparse matrices (Table 14). In general, large matrix problems usually involve sparse matrices and large dense matrix problems are much less common.

Application area	**Dense matrix types**	**Computational problems**
electromagnetics (Helmholtz equation), e.g. radar technology, stealth (i.e. "radar-invisible") airplane technology	complex, Hermitian (rarely also non-Hermitian), e.g. 55296 by 55296	boundary integral solution (specifically the method of moments)
flow analysis (Laplace or Poisson equation), e.g. airflow past an airplane wing, flow around ships	symmetric, e.g. 12088 by 12088	boundary integral solution (specifically the panel method)
diffusion of solid bodies in liquids	block Toeplitz	i. n. a.[156]
diffusion of light through small particles	block Toeplitz	i. n. a.
noise reduction	block Toeplitz	i. n. a.
quantum mechanical scattering (computing the scattering of elementary particles from other particles and atoms; involves Schrödinger wave function)	i. n. a.	dense linear systems
quantum chemistry (Schrödinger wave function)	real symmetric, occasionally Hermitian, small and dense (large systems are usually sparse)	symmetric eigenvalue problems
material science	i. n. a.	unsymmetric eigenvalue problems
real-time signal processing applications	i. n. a.	rank-revealing factorizations and the updating of factorizations after low rank changes

Table 13 *Examples of application areas for dense matrix computations (based on examples found in [Ede91, Ede93, Ede94,Hig96])*

Application area	Sparse matrix types	Computational problems
static analyses in structural engineering[157], e.g. static analysis of buildings, roofs, bridges, airplanes, etc.	real symmetric positive definite, pattern symmetric indefinite, e.g. 3948 by 3948 with 60882 entries	generalized symmetric eigenvalue problem, finite-element modeling, linear systems
dynamic analysis in structural engineering, e.g. dynamic analysis of fluids, suspension bridges, transmission towers, robotic control	real symmetric and positive definite or positive semi-definite or indefinite	symmetric eigenvalue problems, linear systems
hydrodynamics	real unsymmetric, e.g. 100 by 100 with 396 entries	eigenvalues of the Jacobi matrix
oceanic modeling, e.g. models of the shallow waves for the Atlantic and Indian Oceans	real symmetric indefinite, real skew symmetric, e.g. 1919 by 1919 with 4831 entries	finite-difference model
acoustic scattering	complex symmetric	i. n. a.
fluid flow modeling, fluid dynamics, flow in networks	real unsymmetric, symmetric structure, e.g. 511 by 511, 2796 entries and 23560 by 23560 with 484256 entries	iterative and direct methods, eigenvalue and eigenvector problems (in perturbation analysis), Lanczos method
petroleum engineering, e.g. oil recovery, oil reservoir simulation	real unsymmetric, symmetric structure, e.g. 2205 by 2205 with 14133 entries	i. n. a.
electromagnetic field modeling, e.g. integrated circuit applications, power lines	real pattern symmetric indefinite, real pattern symmetric positive definite, real unsymmetric, e.g. 1074 by 1074 with 5760 entries	finite-element modeling, symmetric and unsymmetric eigenvalue problem
power systems simulations, power system networks	real unsymmetric, real symmetric indefinite, real symmetric positive definite, e.g. 4929 by 10595 with 47369 entries	symmetric and unsymmetric eigenvalue problems
circuit simulation	real unsymmetric, 58 by 59 with 340 entries	i. n. a.
astrophysics, e.g. nonlinear radiative transfer and statistical equilibrium in astrophysics	real unsymmetric, e.g. 765 by 765 with 24382 entries	i. n. a.
nuclear physics, plasma physics	real unsymmetric, e.g. 1700 by 1700 with 21313 entries	Large unsymmetric generalized eigenvalue problems
quantum chemistry	complex symmetric indefinite, e.g. 2534 by 2534 with 463360 entries	symmetric eigenvalue problems

chemical engineering, e.g. simple chemical plant model, hydrocarbon separation problem	real unsymmetric, e.g. 225 by 225 with 1308 entries	conjugate gradient eigenvalue computation, initial Jacobian approximation for sparse nonlinear equations
probability theory and its applications, e.g. simulation studies in computer systems involving Markov modeling techniques	real unsymmetric, e.g. 163 by 163 with 935 entries	unsymmetric eigenvalues and eigenvectors
economic modeling e.g. economic models of countries, models of economic transactions	real unsymmetric, e.g. 2529 by 2529 with 90158 entries	i. n. a.
demography, e.g. model of inter-country migration	real unsymmetric, often relatively large fill-in with no pattern, e.g. 3140 by 3140 with 543162 entries	i. n. a.
surveying	real unsymmetric, e.g. 480 by 480 with 17088 entries	least squares problem
air traffic control	sparse real symmetric indefinite, e.g. 2873 by 2873 with 15032 entries	conjugate gradient algorithms
ordinary and partial differential equations	real symmetric positive definite, real symmetric indefinite, real unsymmetric, e.g. 900 by 900 with 4322 entries	symmetric and unsymmetric eigenvalue problems

Table 14 *Examples of application areas for sparse matrix computations (based on examples found in [MM])*

10.1.1.2.4 Existing Matrix Computation Libraries

As of writing, the most comprehensive matrix computation libraries available are written in Fortran. However, several object-oriented matrix computation libraries (for performance reasons, they are written mostly in C++) are currently under development. Table 15 and Table 16 list some of the publicly and commercially available matrix computation libraries in Fortran and in C++ (also see [OONP]).

Matrix computations library	**Features**
LINPACK a matrix computation library for solving dense linear systems; superseded by LAPACK see [DBMS79] and http://www.netlib.org/linpack	*language*: Fortran *matrix types*: dense, real, complex, rectangular, band, symmetric, triangular, and tridiagonal *computations*: factorizations (Cholesky, QR), systems of linear equations (Gaussian elimination, various factorizations), linear

	least squares problems, and singular value problems
EISPACK a matrix computation library for solving dense eigenvalue problems; superseded by LAPACK see [SBD+76] and http://www.netlib.org/eispack	*language*: Fortran *matrix types*: dense, real, complex, rectangular, symmetric, band, and tridiagonal *computations*: eigenvalues and eigenvectors, linear least squares problems
LAPACK a matrix computation library for dense linear problems; supersedes both LINPACK and LAPACK see [ABB+94] and http://www.netlib.org/lapack	*language*: Fortran *matrix types*: dense, real, complex, rectangular, band, symmetric, triangular, and tridiagonal *computations*: systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems
ARPACK a comprehensive library for solving real or complex and symmetric or unsymmetric eigenvalue problems; uses LAPACK and BLAS (see text below Table 16) see [LSY98] and http://www.caam.rice.edu/software/ARPACK	*language*: Fortran *matrix types:* provided by BLAS and LAPACK *computations*: Implicitly Restarted Arnoldi Method (IRAM), Implicitly Restarted Lanczos Method (IRLM), and supporting methods for solving real or complex and symmetric or unsymmetric eigenvalue problems
LAPACK++ a matrix computation library for general dense linear problems; provides a subset of LAPACK functionality in C++ see [DPW93] and http://math.nist.gov/lapack++	*language*: C++ *matrix types*: dense, real, complex, rectangular, symmetric, symmetric positive definite, band, triangular, and tridiagonal *computations*: factorizations (LU, Cholesky, QR), systems of linear equations and eigenvalue problems, and singular value problems
ARPACK++ subset of ARPACK functionality in C++ (using templates) see [FS97] and http://www.caam.rice.edu/software/ARPACK/arpack++.html	*language*: C++ *matrix types*: dense, sparse (CSC), real, complex, rectangular, symmetric, band *computations*: Implicitly Restarted Arnoldi Method (IRAM)
SparseLib++ library with sparse matrices; intended to be used with IML++ see [DLPRJ94] and http://math.nist.gov/sparselib++	*language*: C++ *matrix types*: sparse

IML++ (Iterative Methods Library) library with iterative methods; requires a library implementing matrices, e.g. SparseLib++ see [DLPR96] and http://math.nist.gov/iml++	*language*: C++ *matrix types*: library implementing matrices *computations*: iterative methods for solving both symmetric and unsymmetric linear systems of equations (Richardson Iteration, Chebyshev Iteration, Conjugate Gradient, Conjugate Gradient Squared, BiConjugate Gradient, BiConjugate Gradient Stabilized, Generalized Minimum Residual, Quasi-Minimal Residual Without Lookahead)
Newmat, version 9 a matrix computation library for dense linear problems; does not use C++ templates see http://nz.com/webnz/robert/nzc_nm09.html	*language*: C++ *matrix types*: dense, real, rectangular, diagonal, symmetric, triangular, band *computations*: factorizations (Cholesky, QR, singular value decomposition), eigenvalues of a symmetric matrix, Fast Fourier
TNT (Template Numerical Toolkit) a C++ matrix computation library for linear problems; it has a template-based design; eventually to supersede LAPACK++, SparseLib++, and IMC++; as of writing, with rudimentary functionality see [Poz96] and http://math.nist.gov/tnt	*language*: C++, extensive use of templates *matrix types*: dense, sparse, real, complex, rectangular, symmetric, triangular *computations*: factorizations (LU, Cholesky, QR), systems of linear equations contains an interface to LAPACK
MTL (Matrix Template Library) a C++ matrix library; it has an STL-like template-based design; its goal is to provide only one version of any algorithm and adapt it for various matrices using parameterization and iterators; it implements register blocking using template metaprogramming; it exhibits an excellent performance comparable to tuned Fortran90 code see [SL98a, SL98b] and http://www.lsc.nd.edu/	*language*: C++, extensive use of templates *matrix types*: dense, sparse, real, complex, rectangular, symmetric, band

Table 15 *Some of the publicly available matrix computation libraries and their features*

Matrix computations library	**Features**
Math.h++ C++ vector, matrix, and an array library in one	*language*: C++ *matrix types*: dense, real, complex, rectangular

Rogue Wave Software, Inc., see http://www.roguewave.com/products/math	*computations*: LU factorization, FFT
LAPACK.h++ works on top of Math.h++; offers functionality of the Fortran LAPACK library in C++ Rogue Wave Software, Inc., see http://www.roguewave.com/products/lapack	*language*: C++ *matrix types*: dense (some from Math.h++), sparse, real, complex, symmetric, hermitian, skew-symmetric, band, symmetric band, hermitian band, lower triangular, and upper triangular matrices *computations*: factorizations (LU, QR, SVD, Cholesky, Schur, Hessenberg, complete orthogonal, tridiagonal), real/complex and symmetric/unsymmetric eigenvalue problems
Matrix<LIB> LINPACK and EISPACK functionality in C++ Matlab-like syntax MathTools Ltd, see http://www.mathtools.com	*language*: C++ *matrix types*: dense, real, complex *computations*: factorizations (Cholesky, Hessenberg, LU, QR, QZ, Schur and SVD), solving linear systems, linear least squares problems, eigenvalue/eigenvector problems
ObjectSuite™ C++: IMSL Math Module for C++ a matrix computation library for dense linear problems Visual Numerics, Inc., see http://www.vni.com/products/osuite	*language*: C++ *matrix types*: dense, real, complex, rectangular, symmetric/Hermitian and symmetric/Hermitian positive definite *computations*: factorizations (LU, Cholesky, QR, and Singular Value Decomposition), linear systems, linear least squares problems, eigenvalue and eigenvector problems, two-dimensional FFTs

Table 16 *Some of the commercially available matrix computation libraries and their features*

A set of basic matrix operations and formats for high-performance architectures has been standardized in the form of the *Basic Linear Algebra Subprograms (BLAS).* The operations are organized according to their complexity into three levels: Level-1 BLAS contain operations requiring O(n) of storage for input data and O(n) time of work, e.g. vector/vector operations (see [LHKK79]), Level-2 BLAS contain operations requiring $O(n^2)$ of input and $O(n^2)$ of work, e.g. matrix-vector multiplication (see [DDHH88]), Level-3 BLAS contain operations requiring $O(n^2)$ of input and $O(n^3)$ of work, e.g. matrix-matrix multiplication (see [DDDH90, BLAS97]). There are also Sparse BLAS [CHL+96], which are special BLAS for sparse matrices. The Sparse BLAS standard also defines various sparse storage formats. Different implementations of BLAS are available from http://www.netlib.org/blas/.

10.1.1.2.5 Features of the Domain of Matrix Computation Libraries

From the analysis of application areas and existing matrix computation libraries (Table 13, Table 14, Table 15, and Table 16), we can derive a number of major matrix types and computational method types which are common in the matrix computations practice. They are listed in Table 17 and Table 18, respectively. The types of matrices and computations represent the main features of the domain of matrix computation libraries and can be used to describe the scope of a matrix computation library. The rationale for including each of these features in a concrete matrix

computation library implementation are also given in Table 17 and Table 18. Some features such as dense matrices and factorizations are basic features required by many other features and they should be included in any matrix computation library implementation. Some other features such as complex matrices and methods for computing eigenvalues are — unless directly required by some stakeholders — optional and their implementation may be deferred.

Matrix type	Rationale for inclusion
dense matrices	Dense matrices are ubiquitous in linear algebra computations and are mandatory for any matrix computation library.
sparse matrices	In practice, large linear systems are usually sparse.
real matrices	Real matrices are very common in linear algebra problems.
complex matrices	Complex matrices are less common than real matrices but still very important for a large class of problems.
rectangular, symmetric, diagonal, and triangular matrices	Rectangular, symmetric, diagonal, and triangular matrices are very common in linear algebra problems and are mandatory for any matrix computation library.
band matrices	Band matrices are common in many practical problems, e.g. a large percentage of the matrices found in [MM] are band matrices.
other matrix shapes (e.g. Toeplitz, tridiagonal, symmetric band)	There is a large number of other matrix shapes which are specialized for various problems. In general, providing all possible shapes in a general purpose matrix computation library is not possible since new applications may require new specialized shapes.

Table 17 *Major matrix types and the rationale for their inclusion in the implemented feature set*

Computational methods	Rationale for inclusion
factorizations (decompositions)	Factorizations are needed for direct methods and matrix analysis and are mandatory for any matrix computation library.
direct methods for solving linear systems	Direct methods (e.g. using the LU factorization) are standard methods for solving linear systems.
least squares methods	The least squares approach is concerned with the solution of overdetermined systems of equations. It represents the standard scientific method to reduce the influence of errors when fitting models to given observations.
symmetric and unsymmetric eigenvalue and eigenvector methods	Eigenvalue methods have numerous applications in science and engineering.
iterative methods for linear systems	Iterative methods for linear systems are the methods of choice for some large sparse systems. There are iterative methods for solving linear systems and for computing eigenvalues.

Table 18 *Major matrix computational methods types and the rationale for their inclusion in the implemented feature set*

Figure 137 summarizes the features of a matrix computation library in a feature diagram. The priorities express the typicality rate of the variable features. These typicality rates are informal and are intuitively based on the analysis of application areas and existing matrix computation libraries.

Another important feature of a matrix computation library, which will not be considered here, is its target computer architecture, e.g. hierarchical memory, multiple processors with distributed memory or shared memory, etc.

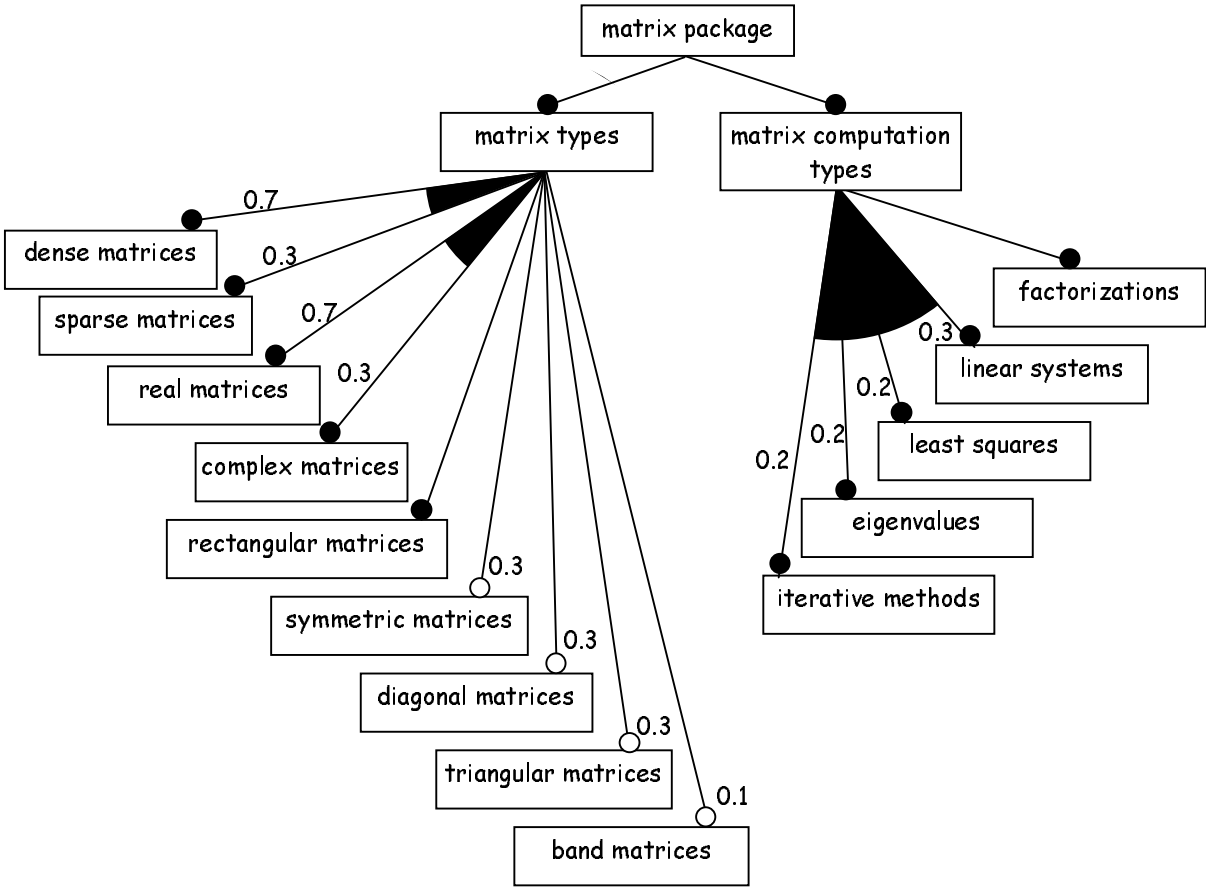

Figure 137 *Feature diagram of a matrix computation library*

10.1.1.2.6 Relationship to Other Domains

The *domain of array libraries* is an example of an *analogy domain* (see Section 3.6.3) of the domain of matrix computation libraries. Array libraries implement arrays (including two-dimensional arrays) and numerical computations on arrays. Thus, there are significant similarities between array libraries and matrix computation libraries. But there are also several differences:

- Array libraries, in contrast to matrix computation libraries, also cover arrays with more than two dimensions.
- Array operations are primarily elementwise operations. For example, in an array library * means elementwise multiply, whereas in a matrix computation library * designates matrix multiplication.
- Arrays usually support a wide range of element types, e.g. int, float, char, bool, and user defined types, whereas the type of matrix elements is either real or complex numbers.
- Array libraries usually do not provide a comprehensive set of algorithms for solving complicated linear problems. They rather focus on other areas, e.g. signal processing.

- Array libraries usually do not provide any special support for different shapes and densities.

Blitz++ [Vel97] is an example of an array library. Math.h++ (see Table 16) combines aspects of both an array and a matrix computation library in one.

An example of another analogy domain is the domain of *image processing libraries*. Images are somewhat similar to matrices. However, there are also several differences:

- The elements of an image are binary, gray scale, or color pixel values. Binary and color pixel values require a different representation than matrix elements. For example, color pixels may be represented using three values and a collection of binary pixels is usually represented by one number.

- The operations and algorithms required in image processing are different than those used for solving linear problems.

An example of a *support domain* is the *domain of container libraries*. A container library, e.g. the Standard Template Library (STL; [MS96, Bre98]) could be used to implement storage for matrix elements in a matrix computation library.

10.1.2 Domain Modeling

10.1.2.1 Key Concepts of the Domain of Matrix Computation Libraries

The key concepts of the domain of matrix computation libraries are

- abstract data types: *vectors* and *matrices*;

- algorithm families: *factorizations, solving systems of linear equations, solving least squares problems, solving eigenvalue and eigenvector problems,* and *iterative methods*.

10.1.2.2 Feature Modeling of the Key Concepts

10.1.2.2.1 Features of Vectors and Matrices

This section describes the features of vectors and matrices. Since the vector features represent a subset of the matrix features, we only list the matrix features and indicate if a feature does not apply to vectors. Please note that vectors can be adequately represented as matrices with number of rows equal one or number of columns equal one.

We have the following matrix features:

- *element type*: type of the matrix elements;

- *subscripts*: subscripts of the matrix elements;

- *structure*: the arrangement and the storage of matrix elements:

- *entry type*: whether an entry is a scalar or a matrix;

- *density*: whether the matrix is sparse or dense;

- *shape*: the arrangement pattern of the nonzero matrix elements (this feature does not apply to vectors);

- *representation*: the data structures used to store the elements;

- *format*: the layout of the elements in the data structures;
- *memory management*: allocating and relinquishing memory;
- *operations*: operations on matrices (including their implementations);
- *attributes*: matrix attributes, e.g. number of rows and columns;
- *concurrency and synchronization*: concurrent execution of algorithms and operations and synchronization of memory access;
- *persistency*: persistent storage of a matrix instance;
- *error handling*: error detection and notification, e.g. bounds checking, compatibility checking for vector-vector, matrix-vector, and matrix-matrix operations.

10.1.2.2.1.1 Element Type

The only element types occurring in linear algebra (and also in the application areas listed in Table 13 and Table 14) are real and complex numbers. Existing libraries (Table 15 and Table 16) typically support single and double precision real and complex element types. Other element types (e.g. bool, user defined types, etc.) are covered by array libraries (see Section 10.1.1.2.6).

10.1.2.2.1.2 Subscripts (Indices)

The following are the subfeatures concerning subscripts:

- *index type*: The type of subscripts is an integral type, e.g. char, short, int, long, unsigned short, unsigned int, or unsigned long.
- *maximum index value*: The choice of index type, e.g. char or unsigned long, determines the maximal size of a matrix or vector.
- *index base*: There are two relevant choices for the start value of indices: *C-style indexing* (or *0-base indexing*), which starts at 0, and the *Fortran-style indexing* (or *1-base indexing*), which starts at 1. Some libraries, e.g. TNT (see Table 15), provide both styles at the same time (TNT provides the operator "[]" for 0-base indexing and the operator "()" for 1-base indexing).
- *subscript ranges*: Additionally, we could also have a subscript type representing subscript ranges. An example of range indexing is the Matlab indexing style [Pra95], e.g. 1:4 denotes a range from 1 to 4, 0:9:3 denotes a range from 0 to 9 with stride 3. An example of a library supporting subscript ranges is Matrix<LIB> (see Table 16).

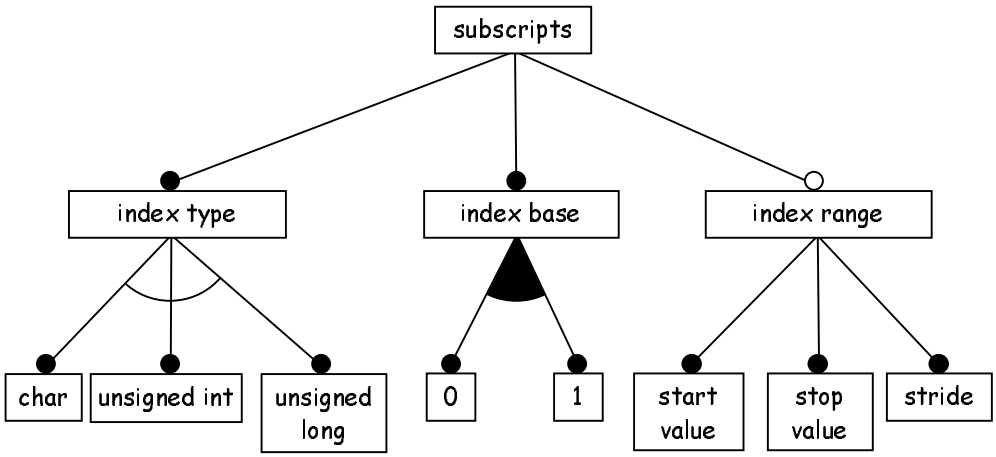

Figure 138 *Subscripts*

10.1.2.2.1.3 Structure

Structure is concerned with the arrangement and the storage of matrix or vector elements. We can exploit the arrangement of the elements in order to reduce storage requirements and to provide specialized and faster variants of basic operations and more complex algorithms. The subfeatures of the *structure* of matrices are shown in Figure 139.

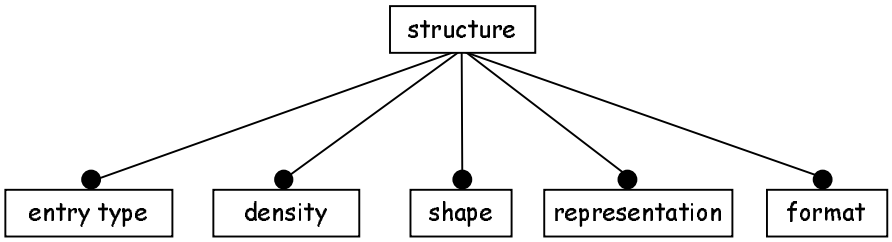

Figure 139 *Structure*

10.1.2.2.1.3.1 Entry Type

The entries in a matrix are usually scalars (e.g. real or complex). These types of matrices are referred to as *point-entry matrices* [CHL+96]. There are also matrices whose entries are matrices and they are referred to as *block matrices* [CHL+96, GL96]. Block matrices are common in high-performance computing since they allow us to express operations on large matrices in terms of operations on small matrices. This formulation enables us to take advantage of the hierarchical memory organization on modern computer architectures.

The memory of modern computer architectures is usually organized into a hierarchy: The higher levels in the hierarchy feature memory fast in access but of limited capacity (e.g. processor cache). As we move down the hierarchy, the memory speed decreases but its capacity increases (e.g. main memory, disk).

When performing a matrix operation, it is advantageous to keep all the operands in cache in order to eliminate excessive data movements between the cache and the main memory during the operation. If the operands are matrices which entirely fit into the cache, we can use the point-entry format. But if a matrix size exceeds the cache size, the block format should be preferred. Operations on block matrices are performed in terms of operations on their blocks, e.g. matrix

multiplication is performed in terms of multiplications of the block submatrices. By properly adjusting the block size, we are able to fit the arguments of the submatrix operation into the cache.

Furthermore, we distinguish between *constant blocks* (i.e. blocks have equal sizes) and *variable blocks* (i.e. blocks have variable sizes). The subfeatures of *entry type* are summarized in Figure 140.

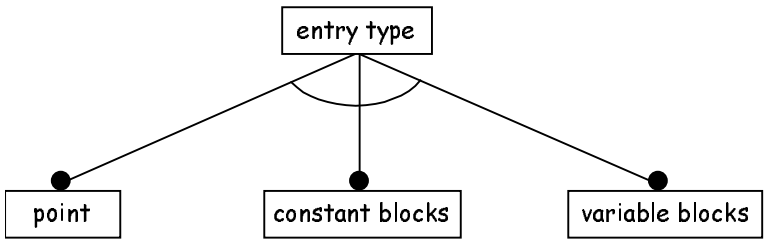

Figure 140 *Entry Type*

Blocking is used in high performance linear algebra libraries such as LAPACK (see Table 15).

10.1.2.2.1.3.2 Density

One of the major distinctions between matrices is whether a matrix is *dense* or *sparse*. A dense matrix is a matrix with a large percentage of *nonzero elements* (i.e. elements not equal zero). A sparse matrix, on the other hand, contains a large percentage of *zero elements* (usually more than 90%). In [Sch89], Schendel gives an example of a sparse matrix in the context of the frequency analysis of linear networks which involves solving a system of linear equations of the form $A(\omega_i)x = b$. In this example, A is a 3304-by-3304 Hermitian matrix with 60685 nonzero elements. Thus, the nonzero elements make up only 0.6% of all elements in A (i.e. the fill-in is 0.6%). Furthermore, the LU-factorization of A yields a new matrix with an even smaller fill-in of 0.4% (see Table 14 for more examples of sparse matrices). The representation of A as a dense matrix would require several megabytes of memory. However, it is necessary to store only the nonzero elements, which dramatically reduces the storage requirements for sparse matrices. The knowledge of the density of a matrix allows us not only to optimize the storage consumption, but also the processing speed since we can provide specialized variants of operations which take advantage of sparseness.

Most matrix computation libraries provide dense matrices and some matrix computation libraries also implement sparse matrices (e.g. LAPACK.h++; see Table 16). Since most of the large matrix problems are sparse (see Table 14), a general-purpose matrix computation library is much more attractive if it implements both dense and sparse matrices.

10.1.2.2.1.3.3 Shape

Matrix computations involve matrices with different arrangement patterns of the nonzero elements. Such arrangement patterns are referred to as *shapes*. Some of the more common shapes include the following (see Figure 141):

- *Rectangular* and *square matrices*: A rectangular matrix has a different number of rows than the number of columns. The number of rows and the number of columns in a square matrix are equal.

- *Null matrix*: A null matrix consists of only zero elements. No elements have to be stored for a null matrix but only the number of rows and columns.

- *Diagonal matrix*: A diagonal matrix is a square matrix with all zero elements except the diagonal elements (i.e. elements whose row index and column index are equal). Only the

(main) diagonal elements have to be stored. If they are all equal, the matrix is referred to as a *scalar matrix* and only the scalar has to be stored.

- *Identity matrix*: An identity matrix is a diagonal matrix whose diagonal entries are all equal 1. No elements have to be stored for an identity matrix but only the number of rows and columns.

- *Symmetric*, *skew-symmetric* or *anti-symmetric*, *Hermitian*, and *skew-Hermitian matrices*: For all elements of a symmetric matrix the following equation holds $a_{ij} = a_{ji}$. For a skew-symmetric matrix, we have a slightly different equation: $a_{ij} = - a_{ji}$. A complex-valued matrix with symmetric real part and skew-symmetric imaginary part is referred to as a Hermitian. If, on the other hand, the real part is skew-symmetric and the imaginary part is symmetric, we have a skew-Hermitian matrix. For all these four matrix types we only need to store one half of the matrix. One possible storage format is to consecutively store all the rows (or columns or diagonals) of one half of the matrix in a vector and use an indexing formula to access the matrix elements.

- *Upper* or *lower triangular* or *unit triangular* or *Hessenberg matrices*: An upper triangular matrix is a square matrix which has nonzero elements only on and above the main diagonal. If the diagonal elements are only ones, the matrix is referred to as *unit upper triangular*. If the diagonal elements are only zeros, the matrix is referred to as *strictly upper triangular*. If, on the other hand, the main diagonal and also the diagonal below contains nonzeros, the matrix is referred to as an upper Hessenberg. The lower triangular, lower unit triangular, and lower Hessenberg matrices are defined analogously. Similarly as in the case of symmetric matrices, only one half of the elements of a triangular matrix has to be stored.

- *Upper* or *lower bidiagonal*, and *tridiagonal matrices*: These matrices are diagonal matrices with an extra nonzero diagonal above, or below, or both above and below the main diagonal.

- *Band matrices*: Band matrices have nonzero fill-in in one or more adjacent diagonals (see Figure 141. Diagonal and triangular matrices can be regarded as a special case of band matrices. An example of a general storage schema for band matrices would be storing the nonzero diagonals in a smaller matrix, with one diagonal per row and accessing the elements using an indexing formula. Special types of band matrices are *upper* and *lower band triangular matrices*, *band diagonal matrices*, and *symmetric band matrices*.

- *Toeplitz matrices*: A Toeplitz matrix is a square matrix, where all elements within each of its diagonals are equal. Thus, a Toeplitz matrix requires the same amount of storage as a diagonal matrix.

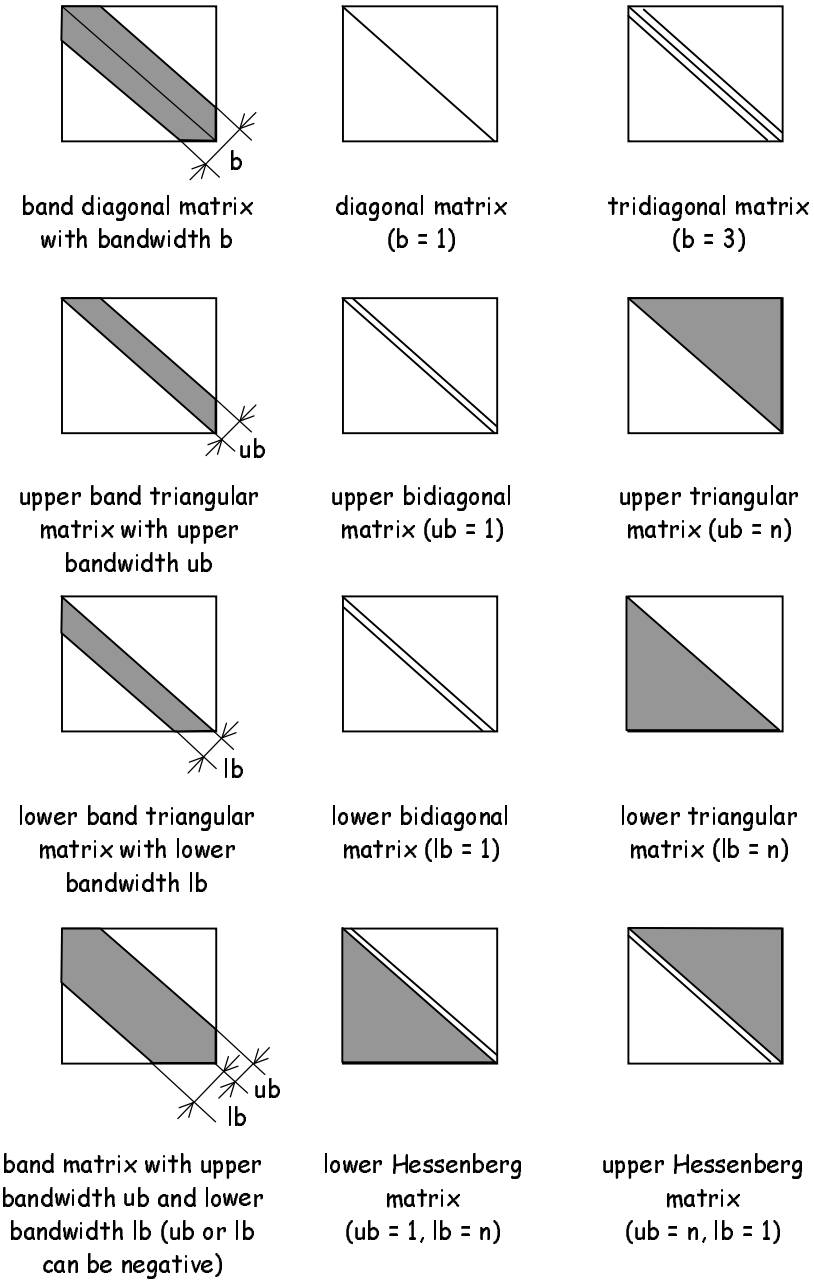

Figure 141 *Examples of n×n band matrices (only the gray region and the shown diagonals may contain nonzeros)*

Some of the above-listed shapes also apply to block matrices, e.g. a block matrix with null matrix entries except for the diagonal entries is referred to as a *block diagonal matrix*. There are also numerous examples of "more exotic", usually sparse matrix types in the literature, e.g. in [Sch89]: *strip matrix, band matrix with margin (also referred to as a bordered matrix), block diagonal matrix with margin, band matrix with step* (see Figure 142).

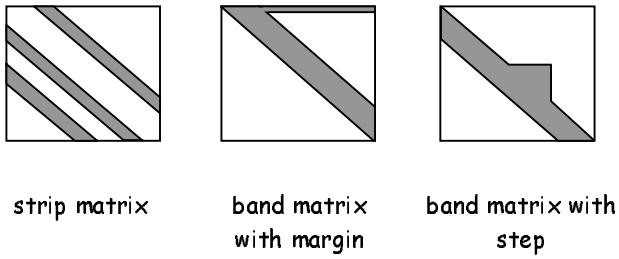

Figure 142 *Some more exotic matrix shapes*

Most matrix computation libraries provide rectangular, symmetric, diagonal, and triangular matrices. Some libraries also provide band matrices (e.g. LAPACK, LAPACK++, ARPACK++, Newmat, LAPACK.h++; see Table 15 and Table 16). Other shapes are less commonly supported.

10.1.2.2.1.3.4 Representation

The elements of a matrix or a vector can be stored in a variety of data structures, e.g. arrays, lists, binary trees, dictionaries (i.e. maps). Each data structure exhibits different performance regarding adding, removing, enumerating, and randomly accessing the elements.

10.1.2.2.1.3.5 Memory Management in Data Structures

The data structures for storing matrix elements may use different memory allocation strategies. We discussed different strategies in Section 9.3.2.2.1. Here, we require at least static and dynamic memory allocation. We extend the representation feature with the *memory allocation* subfeature. The resulting diagram is shown in Figure 143.

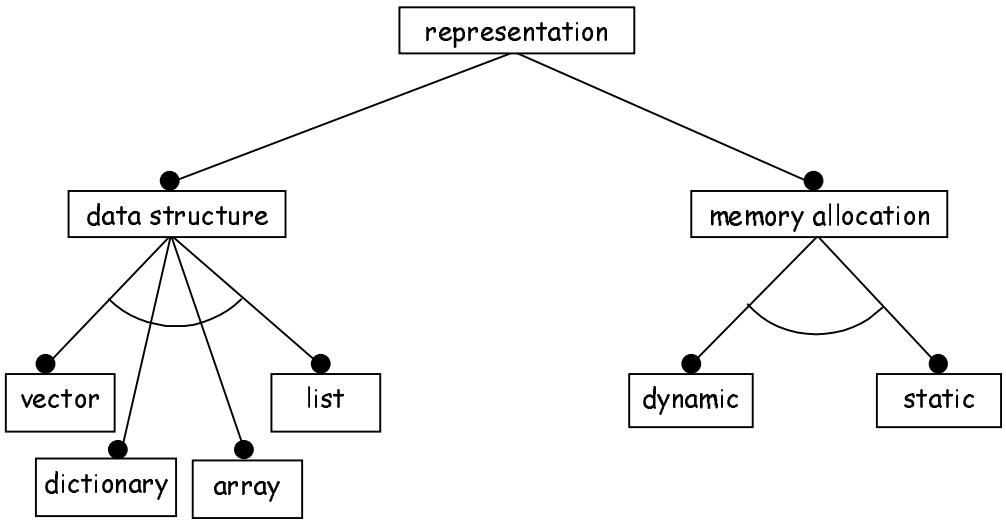

Figure 143 *Representation*

10.1.2.2.1.3.6 Format

Format describes how the elements of a matrix of certain entry type, shape, and density are stored in concrete data structures. For the sake of simplicity, we will further investigate only dense or sparse, point-entry matrices with the most common shapes: rectangular, symmetric, triangular,

diagonal, and band. We first describe the common formats for rectangular dense matrices and general sparse matrices and then discuss the special storage and access requirements of other shapes.

10.1.2.2.1.3.6.1 Rectangular Dense Matrices

There are two major formats for storing the elements of a rectangular dense matrix:

1. *row-major* or *C-style format*: In the row-major format, the matrix elements are stored row-wise, i.e. the neighboring elements of one row are also adjacent in memory. This format corresponds to the way arrays are stored in C.

2. *column-major* or *Fortran-style format*: In the column-major format, the matrix elements are stored column-wise, i.e. the neighboring elements of one column are also adjacent in memory. This format corresponds to the array storage convention of Fortran.

Newer matrix computation libraries usually provide both formats (e.g. LINPACK++, TNT). The column-major format is especially useful for interfacing to Fortran libraries.

10.1.2.2.1.3.6.2 General Sparse Matrices

There are several common storage formats for general sparse matrices, i.e. formats that do not assume any specific shape. However, they are also used to represent shaped sparse matrices. The general sparse storage formats include the following:

- *coordinate format* (COO): Only the nonzero matrix elements along with their coordinates are stored. This format is usually implemented using three vectors, one containing the nonzeros and the other two containing their row and the column indices, respectively. Another possibility is to use one array or list with objects, where each of the objects encapsulates a matrix element and its coordinates. Yet another possibility is to use a hash dictionary data structure, where the keys are the coordinates and the values are the nonzeros.

- *compressed sparse column format* (CSC): The nonzeros are stored column-wise, i.e. the nonzeros of a column are stored in the order of their occurrence within the columns. One possibility is to store the columns containing nonzeros in sparse vectors.

- *compressed sparse row format* (CSR): The nonzeros are stored row-wise, i.e. the nonzeros of a row are stored in the order of their occurrence within the rows. One possibility is to store the rows containing nonzeros in sparse vectors.

There are also several other sparse formats including *sparse diagonal* (DIA), *ellpack/itpack* (ELL), *jagged diagonal* (JAD), and *skyline formats* (SKY) and several *block matrix formats* (see [CHL+96]). Table 19 summarizes when to use which sparse format.

Sparse Format	When to Use?
coordinate (COO)	Most flexible data structure when constructing or modifying a sparse matrix.
compressed sparse column (CSC)	Natural data structure for many common matrix operations including matrix multiplication and constructing or solving sparse triangular factors.
compressed sparse row (CSR)	Natural data structure for many common matrix operations including matrix multiplication and constructing or solving sparse triangular factors.
sparse diagonal (DIA)	Particularly useful for matrices coming from finite difference approximations to partial differential equations on uniform grids.
ellpack/itpack (ELL)	Appropriate for finite element or finite volume approximations to partial differential equations where elements are of the same type, but the gridding is irregular.
jagged diagonal (JAD)	Appropriate for matrices which are highly irregular or for a general-purpose matrix multiplication where the properties of the matrix are not known a priori.
skyline (SKY)	Appropriate for band triangular matrices. Particularly well suited for Cholesky or LU decomposition when no pivoting is required. In this case, all fill will occur within the existing nonzero structure.

Table 19 *Choice of sparse format (adapted from [CHL+96])*

10.1.2.2.1.3.6.3 Dependency Between Density, Shape, and Format

The storage and access requirements of dense or sparse, point-entry matrices with the shapes rectangular, symmetric, triangular, diagonal, and band are set out in Table 20.

Structure Type	Storage and access requirements
dense rectangular	We store the full matrix in the row- or the column-major dense format.
dense symmetric	We store one half of the matrix, e.g. row-, column-, or diagonal-wise in a dense vector, and use an indexing formula to access the elements. Alternatively, we can store the elements in a full-size two-dimensional array using only half of it. The latter approach needs double as much memory as the first one, but it is faster in access since we do not have to transform the indices. Assigning a value to the element a_{ij} with $i \neq j$, automatically assigns the same value to a_{ji}.
dense triangular	We store the nonzero half of the matrix, e.g. row-, column-, or diagonal-wise in a dense vector. Alternatively, we can store the elements in a two-dimensional array, which requires more space but is faster in access. Reading an element from the other half returns 0 and setting such an element to a value other than 0 results in an error.
dense diagonal	We store only the diagonal in a dense vector. Reading an element off the diagonal returns 0 and setting such an element to a value other than 0 results in an error.
dense band	We store the band only, e.g. diagonal-wise in a dense vector or a smaller two-dimensional array. Alternatively, we can store the elements in a full-size two-dimensional array, which requires more space but is faster in access. Reading an element off the band returns 0 and setting such an element to a value other than 0 results in an error.
sparse rectangular	We store only the nonzero elements in one of the sparse formats, e.g. CSR, CSC, COO, ELL, JAD;
sparse symmetric	We store only one half of the matrix and only the nonzero elements using one of the sparse formats (esp. SKY or DIA). Assigning a value to the element a_{ij} with $i \neq j$, automatically assigns the same value to a_{ji}.
sparse triangular	We use one of the sparse formats (esp. SKY or DIA) to store the nonzero elements. Setting an element in the zero-element half of the matrix to a value other than 0 results in an error.
sparse diagonal	We use one of the sparse matrix formats (esp. DIA) to store the nonzero elements or we store them in a sparse vector. Reading elements off the diagonal returns 0 and assigning a value other than 0 to them causes an error.
sparse band	We use one of the sparse formats (esp. DIA for band diagonal and DIA or SKY for band triangular) to store the nonzero elements. Setting an element off band to a value other than 0 results in an error.

Table 20 *Storage and access requirements of matrices of different structures*

The particular shape of a matrix —especially of a sparse matrix —is application dependent and not all of the possible shapes and formats can be provided by a matrix computation library. Thus, it is important to allow a client program to supply specialized formats.

10.1.2.2.1.3.7 Error Checking: Checking Assignment Validity

Checking the validity of an assignment, e.g. checking whether the value assigned to an element within the zero-element half of a triangular matrix is actually a zero, should be parameterized.

10.1.2.2.1.4 Matrix and Vector Operations

We will consider the following operations on matrices and vectors as parts of a matrix component:

- access operations, i.e. set element and get element and
- basic mathematical operations directly based on access operations, e.g. matrix addition and multiplication.

More complex operations, such as computing the inverse of a matrix or solving triangular systems, will be analyzed together with the algorithm families (e.g. solving linear systems).

The basic operations can be clustered according to their arity and argument types. The unary operations are listed in Table 21. The operation type indicates the input argument type and result type. They are separated by an arrow.

Operation type	**Operations**
vector → scalar	Vector norms, e.g. p-norms (1-norm, 2-norm, etc.)
vector → vector	transposition
matrix → scalar	matrix norms, e.g. Frobenius norm, p-norms determinant
matrix → matrix	transposition

Table 21 *Unary operations*

The binary operations are set out in Table 22. An *update* operation stores the result in one of its input arguments. The definitions of the operations listed in Table 21 and Table 22 can be found in [GL96].

Operation type	Operations
(scalar, vector) → vector	scalar-vector multiplication
(scalar, matrix) → matrix	scalar-matrix multiplication
(scalar, vector) → update vector	saxpy, which is defined as follows y := ax + y. where x, y ∈ R^n and a ∈ R
(vector, vector) → vector	vector addition, vector difference, vector multiply (or the Hadamard product)
(vector, vector) → scalar	dot product
(vector, vector) → matrix	outer product
(vector, vector) → update matrix	outer product update, which is defined as follows A := A + xy^T, where x ∈ R^m, y ∈ R^n A ∈ $R^{m \times n}$
(matrix, vector) → vector	matrix-vector multiplication
(matrix, vector) → update vector	gaxpy (i.e. generalized saxpy), which is defined as follows y := Ax + y, where x, y ∈ R^n and A ∈ $R^{m \times n}$
(matrix, matrix) → matrix	matrix addition, matrix difference, matrix multiplication

Table 22 *Binary operations*

A standard set of vector and matrix operations is defined in the form of the BLAS (see Section 10.1.1.2.4). The BLAS are a superset of the operations listed in Table 21 and Table 22. Matrix algorithms can be expressed at different levels, e.g. at the level of operations on matrix elements or at the Level-2 or Level-3 BLAS. Level-3 BLAS formulation of an algorithm contains matrix-matrix operations as their smallest operations. This formulation is especially suited for block matrices.

10.1.2.2.1.4.1 Error Checking: Bounds Checking

Invoking the get or the set operation on a matrix or vector with subscripts which are outside the matrix dimensions or vector dimension is an error. Checking for this condition should be parameterized.

10.1.2.2.1.4.2 Error Checking: Checking Argument Compatibility

The vectors and matrices supplied as input arguments to one of the binary operations must have compatible dimensions. For addition and subtraction of vectors and matrices and dot and outer product, the corresponding dimensions of both arguments must be equal. For the matrix-matrix multiplication, the number of columns of the first matrix must be equal to the number of rows in the second matrix. Similarly, the dimension of the vector in a matrix-vector product must be equal to the number of columns of the matrix. Moreover, a determinant can be computed only for square matrices. Checking argument compatibility should be parameterized. If the numbers of rows and columns are available at compile time, the checking should be performed at compile time.

10.1.2.2.1.5 Interaction Between Operations and Structure

The operations on matrices and vectors interact with their structures in various ways:

- There are dependencies between the shape of the arguments and the shape the result of operations.
- There are dependencies between the density of the arguments and the density of the result of operations.
- The implementation algorithms of the matrix operations can be specialized based on the shape to save floating point operations.
- The implementation of an operation's algorithm depends on the underlying representation and format of the arguments, e.g. dense storage provides fast random access. This is not the case with most sparse storage formats.

The following is true about the shape of the result of an operation:

- the result of multiplying a matrix by a scalar is a matrix of the same shape;
- adding, subtracting, or multiplying two lower triangular matrices results in a lower triangular matrix;
- adding, subtracting, or multiplying two upper triangular matrices results in an upper triangular matrix;
- adding or subtracting two symmetric matrices results in a symmetric matrix;
- adding, subtracting, or multiplying two diagonal matrices results in a diagonal matrix.

When we consider rectangular, triangular, and diagonal matrices, the addition, subtraction, or multiplication of two such matrices can potentially produce a matrix whose shape is equal to the shape resulting from superimposing the shapes of the arguments, e.g. rectangular and diagonal matrices yield rectangular matrices and lower diagonal and upper diagonal matrices also yield rectangular matrices, but diagonal and lower triangular matrices yield lower triangular matrices.

Adding, subtracting, or multiplying two dense matrices results — in most cases — in a dense matrix. Adding or subtracting two sparse matrices results in a sparse matrix. Multiplying two sparse matrices can result in a sparse or a dense matrix.

The algorithms of the matrix operations can be specialized based on the shape of the arguments. For example, the multiplication of two lower triangular matrices requires about half the floating point operations needed to multiply two rectangular matrices. Some of the special cases are adding, subtracting, and multiplying two diagonal matrices, two lower or upper matrices, or a diagonal and a triangular matrix, or multiplying a matrix by a null or identity matrix.

10.1.2.2.1.6 Optimizations

In addition to specializing algorithms for different shapes of the argument matrices, we can also optimize whole expressions. For example, more than one adjacent matrix addition operations in an expression should be all performed using one pair of nested loops adding the matrices elementwise without any intermediate results. Thus, this optimization involves the elimination of temporaries and loop fusing. We already described it in Section 9.4.1.

10.1.2.2.1.7 Attributes

An important attribute of a vector is its *dimension* (or *length*), which is the number of elements the vector contains. Since matrices are two dimensional, they have two attributes describing their size: *number of rows* and *number of columns*. For a square matrix, the number of rows and the

number of columns are equal. Thus, we need to specify only one number, which is referred to as the *order*. For band matrices, we have to specify the *bandwidth* (i.e. the number of nonzero diagonals; see Figure 141). It should be possible to specify all these attributes statically or dynamically.

10.1.2.2.1.8 Concurrency and Synchronization

Matrix operations are well suited for parallelization (see [GL96,p. 256]). However, parallelization of matrix algorithms constitutes a complex area on its own and we will not further investigate this topic. A simple form of concurrent execution, however, can be achieved using threads, i.e. lightweight processes provided by the operating system. In this case, we have to synchronize the concurrent access to shared data structures, e.g. matrix element containers, matrix attributes. This can be achieved through various locking mechanisms, e.g. semaphores or monitors. We use the locking mechanisms to make all operations of a data structure mutually exclusive and to make the writing operations self exclusive (see Section 7.4.3). In the simplest case, we could provide a matrix synchronization wrapper, which makes get and set methods mutually exclusive and the set method self exclusive.

10.1.2.2.1.9 Persistency

We need to provide methods for storing a matrix instance on a disk in some appropriate format and for restoring it back to main memory.

10.1.2.2.2 Matrix Computation Algorithm Families

During Domain Definition in Section 10.1.1.2.5, we identified the main areas of matrix computations:

- factorizations,
- solving linear systems,
- computing least squares solutions,
- eigenvalue computations, and
- iterative methods.

Each of these areas contain large families of matrix computation algorithms.

As an example, we will discuss the family of factorization algorithms. The discussion focuses on the structure of this family rather than on explaining all the mathematical concepts behind the algorithms. The interested reader will find detailed explanations of these concepts in [GL96].

In general, factorizations decompose matrices into factor matrices with some desired properties by applying a number of transformations. Factorizations are used in nearly all the major areas of matrix computations: solving linear systems of equations, computing least squares solutions, and eigenvalue computations. For example, the LU factorization of a matrix A computes the lower triangular matrix L and the upper triangular matrix U, such that A = L*U. The LU factorization can be used to solve a linear system of the form A*x=b, where A is the coefficient matrix, b is the right-hand side vector, and x is the sought-after solution vector. After factoring A into L and U, solving the system involves solving two triangular systems: L*y=b and U*x=y, which is very simple to do using *forward* or *back substitution*.

In general, we can solve a linear system using either factorizations such as the LU, which are also referred to as *direct methods*, or we can use so-called *iterative methods*. Iterative methods generate series of approximate solutions, which hopefully converge on a single solution. Examples of iterative methods for solving linear systems are Jacobi iterations, Gauss-Seidel iterations, SOR iterations, and the Chebyshev semi-iterative method (see [GL96]).

There are two important categories of factorizations: the *LU family* and *orthogonal factorizations*. Examples of the latter are Singular Value Decomposition (SVD) and the QR factorizations (e.g. Hausholder QR, Givens QR, Fast Givens QR, Gram-Schmidt QR).

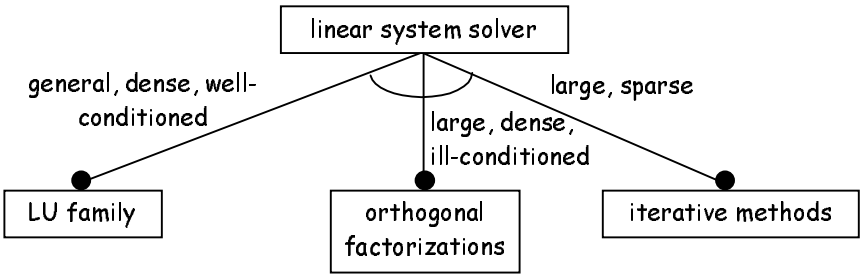

Figure 144 *Approaches to solving linear systems*

Figure 144 explains when to use LU factorizations and when orthogonal factorizations or iterative methods to solve a linear system. Each alternative method is annotated with the properties of the coefficient matrix A as preconditions. These preconditions indicate when a given method is most appropriate. For example, if A is ill-conditioned, LU factorizations should not be used. A is ill-conditioned if it is nearly singular. Whether A is ill-conditioned or not is determined using *condition estimators* (see [GL96] for details).

In the rest of this section, we will concentrate on LU factorizations. The LU factorization in its general form, i.e. *general LU*, corresponds to the Gaussian elimination method. The general LU can be specialized in order to handle systems with special properties more efficiently. For example, if A is square and positive definite, we use the Cholesky factorization, which is a specialization of the general LU.

There are specialized versions of LU factorizations for different matrix shapes, e.g. band matrices or Hessenberg matrices, and for different entry types, i.e. point-entry and block-entry variants (see [GL96]).

An important issue in factorization algorithms is *pivoting*. Conceptually, pivoting involves data movements such as the interchange of two matrix rows (and columns, in some approaches). Gaussian elimination without pivoting fails for a certain class of well-conditioned systems. In this case, we have to use pivoting. However, if pivoting is not necessary, it should be avoided since it degrades performance. We have various pivoting strategies, e.g. no pivoting, partial pivoting, or complete pivoting. Some factorization algorithms have special kinds of pivoting, e.g. symmetric pivoting or diagonal pivoting. In certain cases, e.g. when using the band version of LU factorizations, pivoting destroys the shape of the matrix. This is problematic if we want to factor dense matrices *in place*, i.e. by storing the resulting matrices in the argument matrix. In-place computation is an important optimization technique in matrix computations allowing us to avoid the movement of large amounts of data.

The pivoting code is usually scattered over the base algorithm causing the code tangling problem we discussed in Chapter 7. Thus, pivoting is an example of an aspect in the AOP sense and we need to develop mechanisms for separating the pivoting code from the base algorithm (see e.g. [ILG+97]).

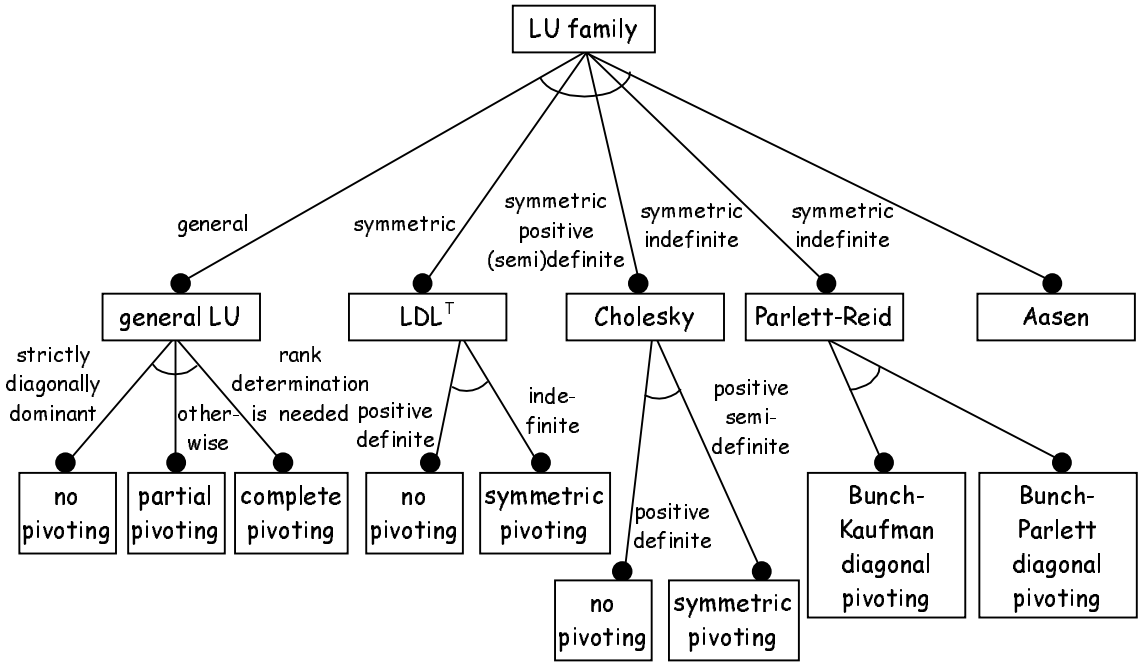

Figure 145 *Feature diagram of the LU family*

A selection of important LU factorizations is shown in Figure 145. The algorithm variants are annotated with matrix properties. An algorithm variant is well-suited for solving a given linear system if the properties of the coefficient matrix of this system match the variant's annotation.

There are also special variants of the LU factorizations for different matrix shapes (not shown in Figure 145). For example, Golub and van Loan describe specializations of general LU, LDL^T, and Cholesky for different band matrices in [GL96]. These specializations work fine without pivoting. Unfortunately, pivoting, when used, destroys the band shape of the factored matrix. As stated, this is problematic if we want to factor dense matrices in place.

In addition to shape, the algorithm selection conditions also include other mathematical properties which are not as easy to determine as shape, e.g.:

- *Positive definite*: Given $A \in R^{n \times n}$, A is positive definite if $x^T*A*x > 0$, for all nonzero $x \in R^n$.
- *Positive semidefinite*: Given $A \in R^{n \times n}$, A is positive definite if $x^T*A*x \geq 0$, for all $x \in R^n$.
- *Indefinite*: Given $A \in R^{n \times n}$, A is indefinite if $A=A^T$ and x^T*A*x takes on both positive and negative values for different $x \in R^n$.

Since all the important properties of a matrix should be encoded in its type, we need to extend the matrix feature model from Section 10.1.2.2.1 with these new mathematical properties. The encoding of these properties in the matrix type allows us to arrange for the automatic selection of the most efficient algorithm variant.

One problem that we will have to address in the implementation is the type mutation in the case of in-place computation, i.e. we want to store the results in the arguments, but the results have different properties than the arguments. One possibility to address this problem is to divide a matrix into an element container and a matrix wrapper which encodes the matrix properties (see [BN94, pp. 459-464]). Given this arrangement, we can do the storing at the element container

level, then mark the old matrix wrapper as invalid, and use a new one, which encodes the new matrix properties.

10.2 Domain Design

Each ADT and each algorithm family should be implemented in a separate component. In the rest of this chapter, we only present the detailed design and implementation of the matrix component.

10.2.1 Architecture of the Matrix Component

Before we discuss the architecture of the matrix component, we first take a look at a concrete example demonstrating how to use it. For our example, we have chosen the C++ implementation although later in Section 10.3.2 you will see an alternative implementation of the matrix component in the IP System. Here is the example:

```
//define a general rectangular matrix with element type double.
typedef MATRIX_GENERATOR<
        matrix< double,
                structure<  rect<>
                          >
              >
>::RET RectMatrixType;

//define a scalar matrix with 3 rows and 3 columns
//scalar value is 3.4
typedef MATRIX_GENERATOR<
        matrix< double,
                structure< scalar< stat_val<int_number<int, 3> >,
                                   stat_val<float_number<double, 3400> >
                                 >
                          >
              >
>::RET ScalarMatrixType;

//declare some matrices
RectMatrixType RectMatrix1(3, 3), RectMatrix2(3, 3);
ScalarMatrixType ScalarMatrix;

//initialization of a dense matrix
RectMatrix1=
        1, 2, 3,
        4, 5, 6,
        7, 8, 9;

//multiplication of two matrices
RectMatrix2= ScalarMatrix * (RectMatrix1+ ScalarMatrix);
```

The first two gray regions indicate *two matrix configuration expressions* and the last one a *matrix expression*. The two kinds of expressions represent two important interfaces to the matrix component:

- *Matrix Configuration DSL Interface*: Configuration expressions are used to define concrete matrix types. The structure of a configuration expressions is described by the Matrix Configuration DSL (MCDSL).

- *Matrix Expression DSL Interface*: Matrix expressions are expressions involving matrices and operations on them. The structure of a matrix expressions is described by the Matrix Expression DSL (MEDSL).

Figure 146 shows the high-level architecture of the matrix component. The matrix configuration expressions are compiled by the *MCDSL generator* and the matrix expressions are compiled by the *MEDSL generator*. The MCDSL generator translates a matrix configuration expression into a

matrix type by composing a number of *implementation components* (*ICs*). The MEDSL generator translates a matrix expression into an efficient implementation, e.g. by composing code fragments.

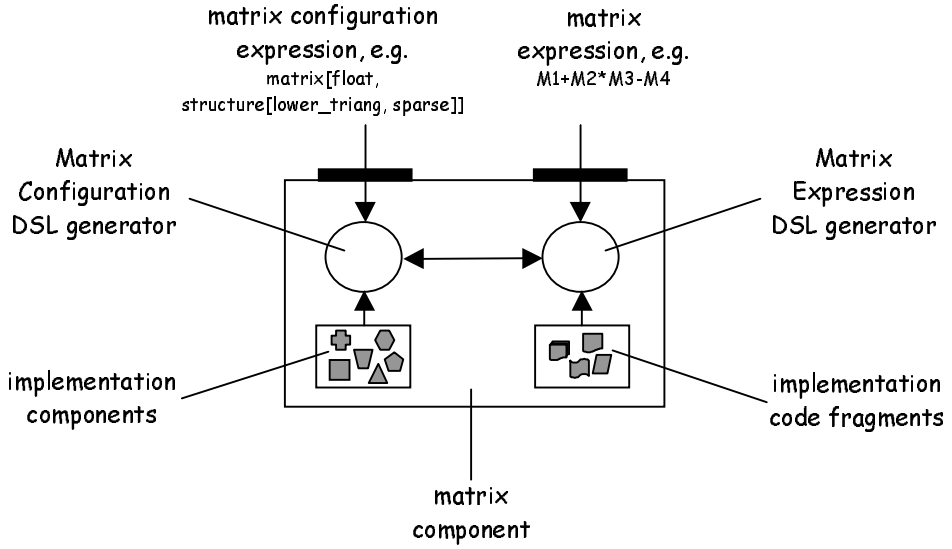

Figure 146 *High-level Architecture of the Generative Matrix Component*

A more detailed view of the matrix component architecture is given in Figure 147. The pipeline on the left corresponds to the MCDSL generator and compiles matrix configuration expressions. The pipeline on the right corresponds to the MEDSL generator and compiles matrix expressions. A matrix configuration expression is compiled by parsing it (which retrieves the values of the features explicitly specified in the configuration expression), assigning default values to the unspecified features (some of which are computed), and assembling the implementation components according to the values of the features into a concrete matrix type. The matrix type also includes a *configuration repository* containing the value of all its configuration DSL features and some other types. The implementation components can be composed only into some valid configurations. These are specified by the *Implementation Components Configuration Language* (*ICCL*). In effect, the MCDSL generator has to translate matrix configuration expressions into corresponding ICCL expressions.

A matrix expression is parsed and then typed by computing the type records of all subexpressions (which requires accessing the configuration repositories of the argument matrices), and finally the efficient implementation code is generated.

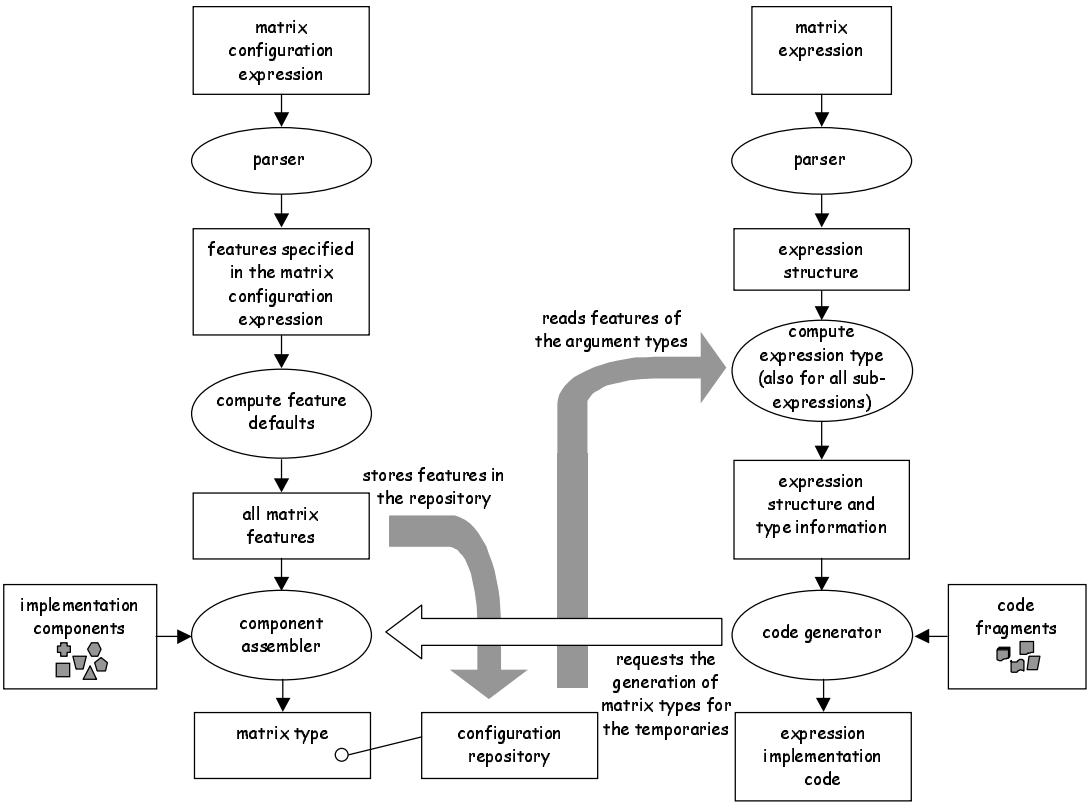

Figure 147 *More detailed architecture view of the matrix component*

The following sections contain the design specifications for

1. Matrix Configuration DSL (Section 10.2.3);
2. Matrix Implementation Components and the Matrix ICCL (Section 10.2.4);
3. Compiling Matrix Configuration DSL into ICCL (Section 10.2.5);
4. Matrix Expression DSL and Optimizing Matrix Expressions (Section 10.2.6);
5. Computing Result Type of the Expressions (Section 10.2.7).

Before we start with the Matrix Configuration DSL, we first need to define the scope of the matrix feature model that we are going to implement.

10.2.2 Matrix Component Scoping

The matrix component will cover a subset of the feature model we came up with in Section 10.1.2.2.1. In particular, we made the following decisions:

- *Element type*: We only support real numbers. Complex numbers are not supported, but we can add them later.
- *Subscripts*: Currently, we only support C-style indexing. Any integral type can be used as index type.

- *Entry type*: We only support point-entry matrices. Blocking can be added for dense matrices using blocking iterators, i.e. iterators that return matrices on dereferencing. Blocking iterators are described in [SL98a]. Sparse matrices would require special blocked formats (see [CHL+96]).

- *Density*: We support both dense and sparse matrices.

- *Shapes*: Currently we support rectangular, scalar, diagonal, triangular, symmetric, band diagonal, and band triangular matrices. Other shapes can be easily added.

- *Formats*: Dense matrices are stored in an array (row- or column-wise) or alternatively in a vector (diagonal-wise). The supported sparse formats include CSR, CSC, COO, DIA, and SKY.

- *Representation*: The elements are stored in dynamic or static, one or two dimensional containers. Other containers could also be easily integrated using adapters.

- *Error checking*: We provide optional bounds checking, compatibility checking, and memory allocation error checking.

- *Concurrency and synchronization*: Concurrency and synchronization are currently not supported.

- *Persistency*: We provide a way to write a matrix to a file in an uncompressed ASCII format and to read it back in.

- *Operations*: We only support matrix-matrix addition, subtraction, and multiplication.

10.2.3 Matrix Configuration DSL

The Matrix Configuration DSL is used to specify the features of a matrix configuration. We designed it according to the strategies described in Section 9.4.3. In particular, the DSL should allow the programmer to formulate matrix specifications at a level of detail which is most suitable for the particular client code. We address this goal by providing direct and computed feature defaults. If a configuration specification does not specify a feature for which we have a direct default, the direct default is assumed. The remaining features are computed from the specified features and other defaults. Furthermore, we introduced two new abstract features: *optimization flag* and *error checking flag*. The possible values of the optimization flag are *speed* and *space* and they allow us to specify whether the matrix should be optimized for speed or space. The error checking flag is used to specify the default for other error checking features. Thus, if we set the error checking flag to *check for errors*, the assumed default for bounds checking, compatibility checking, and memory allocation error checking will be to check for errors. Of course, each of the individual error checking features may be explicitly specified to have another value.

The Matrix Configuration DSL uses parameterization as its only variability mechanism, i.e. it only contains mandatory and alternative features. The reason for this is that we want to be able to represent it using C++ templates. We can always convert an arbitrary feature diagram into one that uses dimensions as its only kind of variability, although the resulting diagram will usually be more complex. If the implementation technology does not limit us to parameterization, we can represent a configuration DSL using the full feature diagram notation. For example, we could implement a GUI to specify configurations using feature diagrams directly or by menus or design some new configuration language.

The Matrix Configuration DSL specification consists of four parts:

- grammar specification,

- description of the features,
- specification of the direct feature defaults, and
- specification of the computation procedures for the computed feature defaults.

The sample grammar in Figure 148 demonstrates the notation we will use later to specify the Matrix Configuration DSL grammar. It describes a very simple matrix concept and is equivalent to the feature diagram in Figure 149. We use brackets to enclose *parameters* (i.e. dimensions) and a vertical bar to separate *alternative parameter values*. The symbols on the right are the *nonterminal symbols*. The first nonterminal (i.e. Matrix) represents the concept. The remaining nonterminals are the parameters (i.e. ElementType, Shape, Format, ArrayOrder, OptimizationFlag). As you may remember from Section 6.4.2, we used this kind of grammars to specify GenVoca architectures.

```
Matrix:           matrix[ElementType, Shape, Format, OptimizationFlag]
ElementType:      real | complex
Shape:            rectangular | lowerTriangular | upperTriangular
Format:           array[ArrayOrder] | vector
ArrayOrder:       cLike | fortranLike
OptimizationFlag: speed | space
```

Figure 148 *Sample configuration DSL grammar*

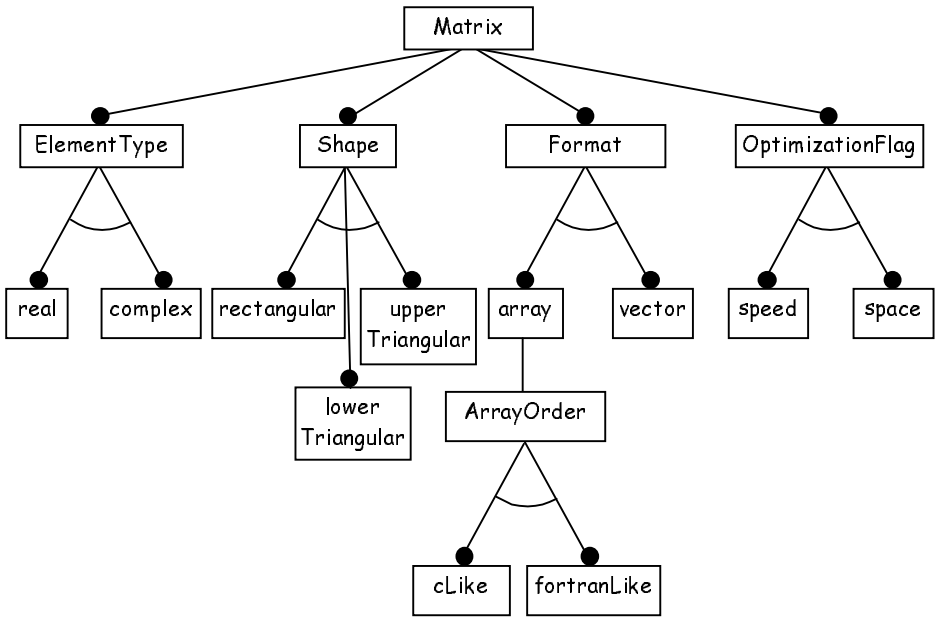

Figure 149 *Feature diagram equivalent to the sample grammar in Figure 148*

Our sample grammar defines 2*3*(2+1)*2 = 36 valid configuration expressions:

```
matrix[real,rectangular,array[cLike],speed]
matrix[real,rectangular,array[cLike],space]
matrix[real,rectangular,array[fortranLike],speed]
matrix[real,rectangular,array[fortranLike],space]
matrix[real,rectangular,vector,speed]
matrix[real,rectangular,vector,space]
matrix[real,lowerTriangular,array[cLike],speed]
```

matrix[real,lowerTriangular,array[cLike],space]
matrix[real,lowerTriangular,array[fortranLike],speed]
matrix[real,lowerTriangular,array[fortranLike],space]
matrix[real,lowerTriangular,vector,speed]
matrix[real,lowerTriangular,vector,space]
matrix[real,upperTriangular,array[cLike],speed]
matrix[real,upperTriangular,array[cLike],space]
matrix[real,upperTriangular,array[fortranLike],speed]
matrix[real,upperTriangular,array[fortranLike],space]
matrix[real,upperTriangular,vector,speed]
matrix[real,upperTriangular,vector,space]
matrix[complex,rectangular,array[cLike],speed]
matrix[complex,rectangular,array[cLike],space]
matrix[complex,rectangular,array[fortranLike],speed]
matrix[complex,rectangular,array[fortranLike],space]
matrix[complex,rectangular,vector,speed]
matrix[complex,rectangular,vector,space]
matrix[complex,lowerTriangular,array[cLike],speed]
matrix[complex,lowerTriangular,array[cLike],space]
matrix[complex,lowerTriangular,array[fortranLike],speed]
matrix[complex,lowerTriangular,array[fortranLike],space]
matrix[complex,lowerTriangular,vector,speed]
matrix[complex,lowerTriangular,vector,space]
matrix[complex,upperTriangular,array[cLike],speed]
matrix[complex,upperTriangular,array[cLike],space]
matrix[complex,upperTriangular,array[fortranLike],speed]
matrix[complex,upperTriangular,array[fortranLike],space]
matrix[complex,upperTriangular,vector,speed]
matrix[complex,upperTriangular,vector,space]

Next, we assume that the parameters of our simple grammar have the direct defaults specified in Table 23. In general, there is usually no such thing as the "absolute" choice for a default value. We often have a strong feeling about some defaults and some others are quite arbitrary. For example, we choose rectangular for Shape since a rectangular matrix can also hold a triangular matrix. This reflects a common principle: We usually choose the most general value to be the default value. This principle, however, does not have to be followed in all cases. Even if complex is more general than real, we still select real as the default value for ElementType. This is so since real reflects the more common case. Of course, what is the more common case and what not depends on the context, our knowledge, etc. Finally, we could have features for which no reasonable defaults can be assumed at all, e.g. the static number of rows and columns of a matrix. These features must be specified explicitly, otherwise we have an error.

ElementType:	real
Shape:	rectangular
ArrayOrder:	cLike
OptimizationFlag:	space

Table 23 *Direct feature defaults for the sample grammar in Figure 148*

Please note that we did not specify a direct default for Format. The reason is that we can compute it based on Shape and OptimizationFlag. The default will be computed as follows. If the value of Shape is rectangular and Format is not specified, we assume Format to be array since (dense) rectangular matrices are optimally stored in an array. If the value of Shape is triangular and Format is not specified, the value of Format depends on OptimizationFlag. If OptimizationFlag is space, we assume Format to be vector since the vector format stores only the nonzero half of the matrix (e.g. diagonal-wise). In this case, the element access functions have to convert the two-dimensional subscripts of a matrix into the one-dimensional subscripts of a vector. If, on the other hand, OptimizationFlag is speed, we assume Format to be array. Storing a triangular matrix in an array wastes space but it allows a faster element access since we

do not have to convert indices. This dependency between Shape, OptimizationFlag, and Format is specified in Table 24.

Shape	OptimizationFlag	Format
rectangular	*	array
lowerTriangular	speed	array
upperTriangular	space	vector

Table 24 *Computing the DSL feature Format for the sample grammar in Figure 148*

We will use tables similar to Table 24 for specifying any dependencies between features. We refer to them as *feature dependency tables*.

Each dependency table represents a function. The columns to the left of the double vertical divider specify the arguments and the columns to the right specify the corresponding result values. For example, Table 25 specifies how to compute the product of two numbers. A dependency table is evaluated row-wise from top to bottom. Given some concrete Factor1 and Factor2, you try to match them to the values specified in the first row. If they match, you terminate the search and take the result from the last corresponding Product cell. If they do not match, you proceed with the next row.

The argument cells of a dependency table may contain one of the following:

- one or more concrete values for the corresponding variable; multiple values are interpreted as alternatives;
- "*", which matches any value;
- a local variable, e.g. (factor); local variables are enclosed in parentheses and denote the current value of the argument;
- "---", which indicates that the corresponding argument does not apply to this row; in terms of matching, it is equivalent to "*";

The result cells may contain one of the following:

- a concrete value,
- an expression; an expression starts with "=" and can refer to the table arguments and local variables;

Factor1	Factor2	Product
0	*	0
*	0	0
*	1	= Factor1
1	*	= Factor2
(factor)	(factor)	=(factor)^2
25	75	1875
*	*	= Factor1*Factor2

Table 25 *Sample specification of the product of two factors [Neu98]*

Now there is the question how the feature defaults are used. Feature defaults allow us to leave out some features in a configuration expression. For example, we could specify a matrix as simply as

```
matrix[]
```

Given the defaults specified above, this expression is equivalent to

```
matrix[real,rectangular,array[cLike],space]
```

How did we come up with this expression? This is simple. We took the default values for ElementType, Shape, and OptimizationFlag directly from Table 23 and then we determined Format based on Table 24. Finally, we took the value for ArrayOrder from Table 23. Other examples are shown in Table 26. Please note that we can leave out one or more parameters of a parameter list only if they constitute the last *n* parameters in the list (this corresponds to the way parameter defaults are used in C++ class templates).

Abbreviated expression	Equivalent, fully expanded expression
matrix[complex]	matrix[complex,rectangular,array[cLike],space]
matrix[real,lowerTriangular]	matrix[real,lowerTriangular, vector,space]
matrix[real,lowerTriangular, array[]]	matrix[real,lowerTriangular,array[cLike],space]
matrix[real,rectangular, vector]	matrix[real,rectangular, vector,space]

Table 26 *Examples of expressions abbreviated by leaving out trailing parameters*

There is also a method for not specifying a parameter in the middle of a parameter list: we use the value unspecified. Some examples are shown in Table 27.

Expression with unspecified	Equivalent, fully expanded expression
matrix[complex,upperTriangular,unspecified,speed]	matrix[complex,upperTriangular,array[cLike],speed]
matrix[complex,unspecified,unspecified,speed]	matrix[complex,rectangular,array[cLike],speed]

Table 27 *Examples of expressions with unspecified values*

10.2.3.1 Grammar of the Matrix Configuration DSL

The grammar of the Matrix Specification DSL is shown in Figure 150. All the features are explained in the following section.

Matrix:	matrix[ElementType, Structure, OptFlag, ErrFlag, BoundsChecking, CompatChecking, IndexType]
ElementType:	float | double | long double | short | int | long | unsigned short | unsigned int | unsigned long
Structure:	structure[Shape, Density, Malloc]
Shape:	rect[Rows, Cols, RectFormat] | diag[Order] | scalar[Order, ScalarValue] | ident[Order] | zero[Order] | lowerTriang[Order, LowerTriangFormat] | upperTriang[Order, UpperTriangFormat] | symm[Order, SymmFormat] | bandDiag[Order, Diags, BandDiagFormat] | lowerBandTriang[Order, Diags, LowerBandTriangFormat] | upperBandTriang[Order, Diags, UpperBandTriangFormat]
RectFormat:	array[ArrOrder] | CSR | CSC | COO[DictFormat]
LowerTriangFormat:	vector | array[ArrOrder] | DIA | SKY
UpperTriangFormat:	vector | array[ArrOrder] | DIA | SKY
SymmFormat:	vector | array[ArrOrder] | DIA | SKY
BandDiagFormat:	vector | array[ArrOrder] | DIA
LowerBandTriangFormat:	vector | DIA | SKY
UpperBandTriangFormat:	vector | DIA | SKY
ArrOrder:	cLike, fortranLike
DictFormat:	hashDictionary[HashWidth] | listDictionary
Density:	dense | sparse[Ratio, Growing]
Malloc:	fix[Size] | dyn[MallocErrChecking]
MallocErrChecking:	checkMallocErr | noMallocErrChecking
OptFlag:	speed | space
ErrFlag:	checkAsDefault | noChecking
BoundsChecking:	checkBounds | noBoundsChecking
CompatChecking:	checkCompat | noCompatChecking
IndexType:	char | short | int | long | unsigned char | unsigned short | unsigned int | unsigned long | signed char
Rows:	statVal[RowsNumber] | dynVal
Cols:	statVal[ColsNumber] | dynVal
Order:	statVal[OrderNumber] | dynVal
Diags:	statVal[DiagsNumber] | dynVal
ScalarValue:	statVal[ScalarValueNumber] | dynVal
Ratio, Growing:	float_number[Type, Value]
RowsNumber, ColsNumber, OrderNumber, DiagsNumber, Size, HashWidth:	int_number[Type, Value]
ScalarValueNumber:	float_number[Type, Value] | int_number[Type, Value]

Figure 150 *Grammar of the Matrix Configuration DSL*

10.2.3.2 Description of the Features of the Matrix Configuration DSL

Figure 151 through Figure 166 show the entire feature diagram for the Matrix Configuration DSL. Each single diagram covers some part of the complete feature diagram. The partial diagrams are followed by tables explaining the features they contain. Each feature has a traceability link back to the section describing its purpose.

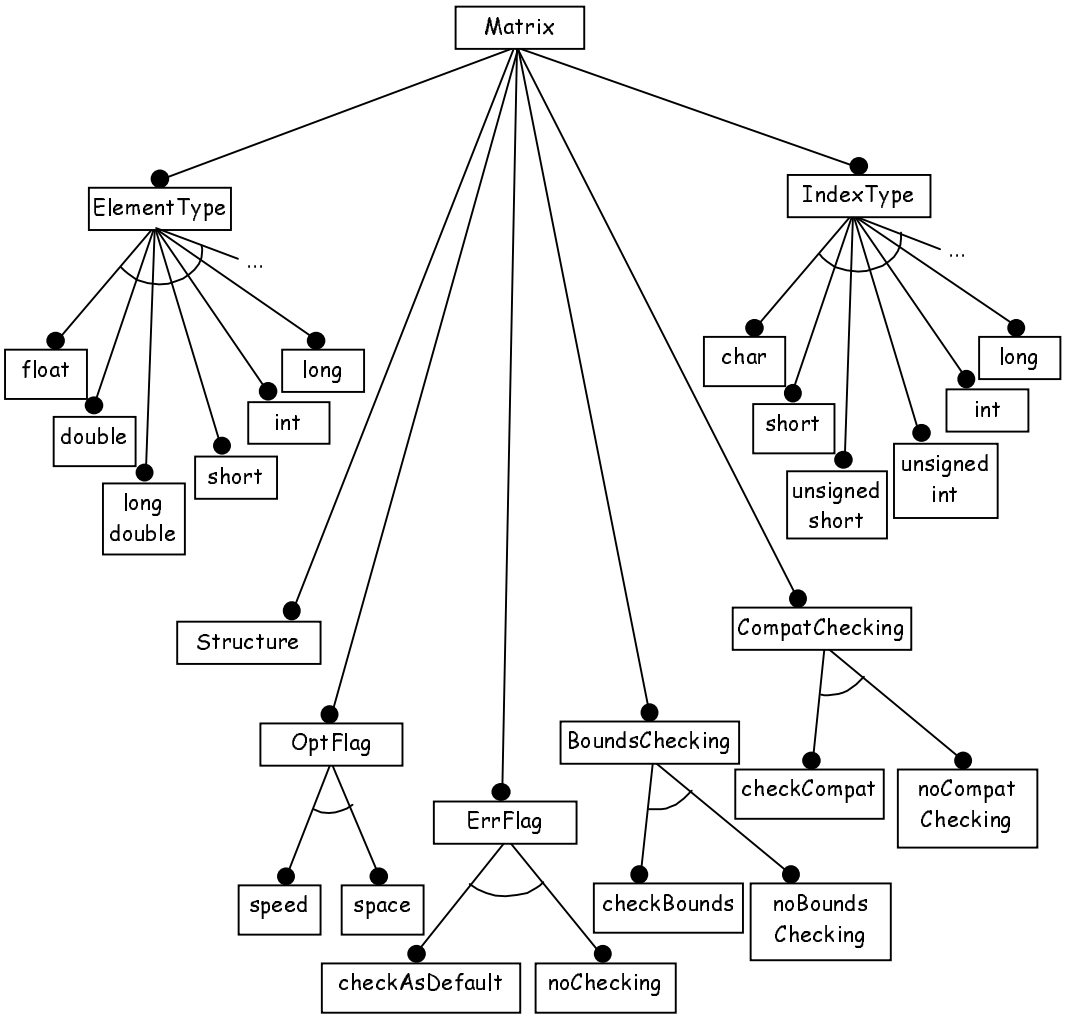

Figure 151 *Matrix features (see Table 28 for explanations)*

Feature name	Description	
ElementType	Type of the matrix elements (see Section 10.1.2.2.1.1).	
	Subfeatures	Possible values of *ElementType* include *float*, *double*, *long double*, *short*, *int*, *long*, *unsigned short*, *unsigned int*, and *unsigned long*. Other number types, if supported on the target platform, are also possible. Note: *complex* currently not supported
Structure	Structure describes shape, density, and memory allocation strategy (but also, indirectly, format and representation) of a matrix (see Section 10.1.2.2.1.3).	
	Subfeatures	see Figure 152
OptFlag	Optimization flag indicating whether the matrix should be optimized for speed or space (see 10.2.3).	
	Subfeatures	Possible values of *OptFlag*: *speed*, *space*
ErrFlag	Error checking flag determines whether all error checking should be done or no checking by default (see 10.2.3). The default is used for a specific error checking feature (e.g. *BoundsChecking* or *CompatChecking*) only if the feature is not specified by the user. For example, if *BoundsChecking* is not specified and *ErrFlag* is *checkAsDefault*, then bounds checking is *checkBounds*.	
	Subfeatures	Possible values of *ErrFlag*: *checkAsDefault*, *noChecking*
BoundsChecking	Bounds checking flag determines whether the validity of the indices used in each access operation to the matrix elements is checked or not (see Section 10.1.2.2.1.4.1).	
	Subfeatures	Possible values of *BoundsChecking*: *checkBounds*, *noBoundsChecking*
CompatChecking	Compatibility checking flag determines whether the compatibility of sizes of the arguments to an operation is checked or not (see Section 10.1.2.2.1.4.2). (For example, two matrices are compatible for multiplication if the number of rows of the first matrix is equal to the number of columns of the second matrix.)	
	Subfeatures	Possible values of *CompatChecking*: *checkCompat*, *noCompatChecking*
IndexType	Type of the index used to address matrix elements (see Section 10.1.2.2.1.2).	
	Subfeatures	Possible values of *IndexType* include *char*, *short*, *int*, *long*, *unsigned char*, *unsigned short*, *unsigned int*, *unsigned long*, and *signed char*. Other number types, if supported on the target platform, are also possible.

Table 28 *Description of matrix features*

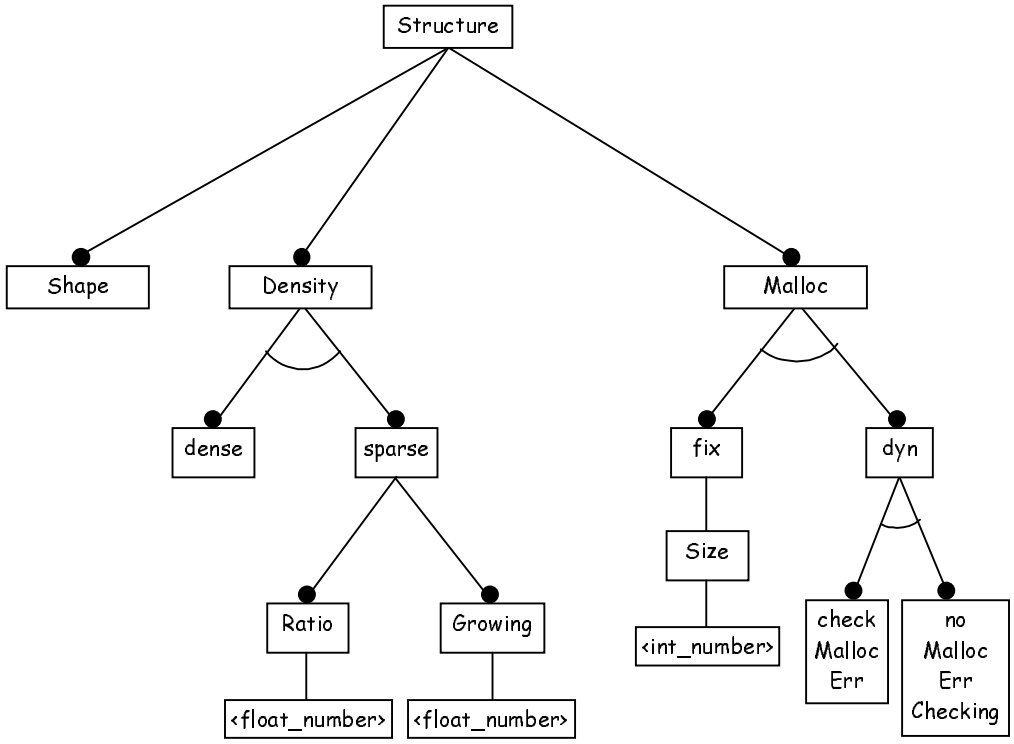

Figure 152 *Subfeatures of Structure (see Table 29 for explanations)*

Feature name	Description	
Shape	Shape describes matrix shape, but also, indirectly, format and representation (see Section 10.1.2.2.1.3.3).	
	Subfeatures	see Figure 153
Density	Density specifies whether a matrix is sparse or dense (see Section 10.1.2.2.1.3.2).	
	Subfeatures	Possible values of *Density*: *sparse*, *dense* *sparse* has two additional subfeatures: *Ratio* and *Growing*. *Ratio* specifies the estimated number of nonzero elements divided by the total number of elements of a matrix, i.e. it is a number between 0 and 1. *Growing* ratio specifies the relative density growth. The density of a matrix grows when nonzero numbers are assigned to zero elements or it decreases when zero is assigned to nonzero elements. *Growing* specifies the relative density change per time unit and it is a float number between 0 and +∞. 0 means no change. 1 means the doubling of the number of the nonzero elements. For example, *Growing* 1 means that the number of nonzero elements grows by factor 4 over two time units. In the matrix implementation, a time unit is the time between points at which the element container allocates extra memory.
Malloc	Memory allocation strategy for the matrix element container (see Section 10.1.2.2.1.3.5).	
	Subfeatures	Possible values of *Malloc*: *fix*, *dyn* *fix* implies static allocation. *dyn* implies dynamic allocation. *fix* has *Size* as its subfeature. *Size* specifies the size of the memory block which is statically allocated for the matrix elements. *Size* specifies only one dimension, i.e. the actual size of the allocated memory block is *Size* *size_of(*ElementType*) for 1D containers and *Size* * *Size* * size_of(*ElementType*) for 2D containers. If the number of rows and the number of columns are specified statically, *Size* is ignored. *dyn* has two alternative subfeatures: *checkMallocErr*, *noMallocChecking*. *checkMallocErr* implies checking for memory allocation errors. *noMallocChecking* implies no checking.

Table 29 *Description of subfeatures of Structure*

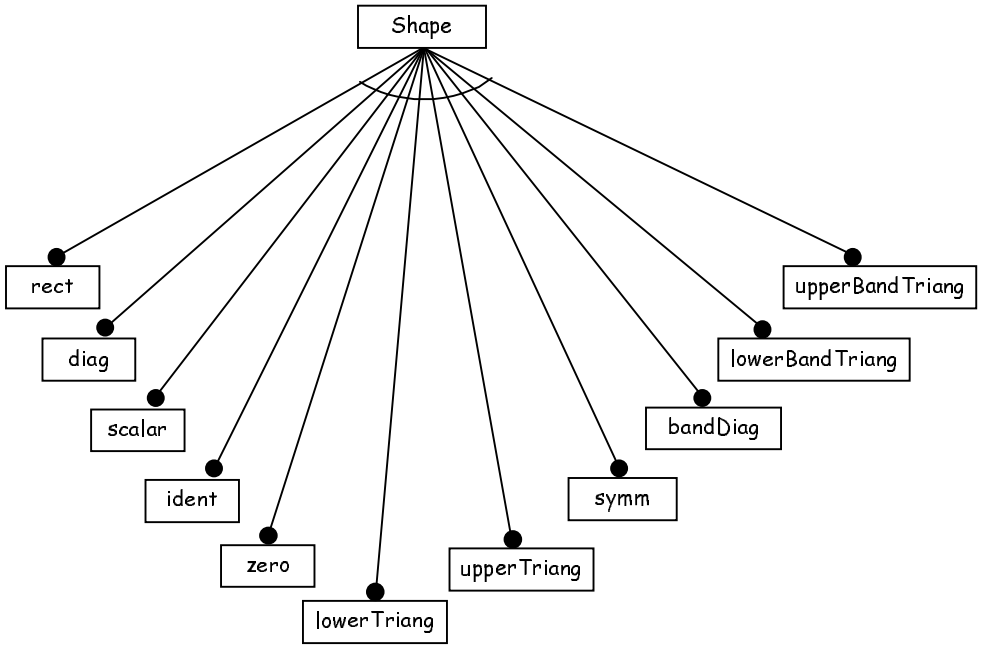

Figure 153 *Subfeatures of Shape (see Table 30 for explanations)*

Feature name	Description
rect	Rectangular matrix (see Figure 154 for subfeatures).
diag	Diagonal matrix (see Figure 155 for subfeatures).
scalar	Scalar matrix (see Figure 156 for subfeatures).
ident	Identity matrix (see Figure 157 for subfeatures).
zero	Zero matrix (see Figure 158 for subfeatures).
lowerTriang	Lower triangular matrix (see Figure 159 for subfeatures).
upperTriang	Upper triangular matrix (see Figure 160 for subfeatures).
symm	Symmetric matrix (see Figure 161 for subfeatures).
bandDiag	Band diagonal matrix (see Figure 162 for subfeatures).
lowerBandTriang	Lower band triangular matrix (see Figure 163 for subfeatures).
upperBandTriang	Upper band triangular matrix (see Figure 164 for subfeatures).

Table 30 *Description of subfeatures of Shape (see Section 10.1.2.2.1.3.3 for explanations)*

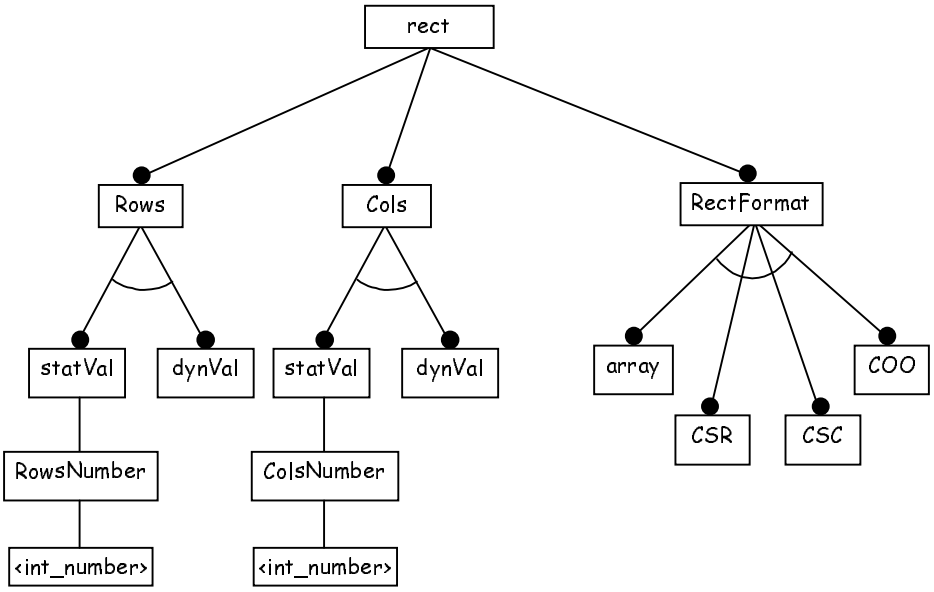

Figure 154 *Subfeatures of rect (see Table 31 for explanations)*

Feature name	Description	
Rows	*Rows* allows us to specify the number of rows of a matrix statically or to indicate that the number can be set at runtime (see Section 10.1.2.2.1.7).	
	Subfeatures	Possible values of *Rows*: *statVal*, *dynVal* *statVal* indicates that the number of rows is specified statically. The subfeature *RowsNumber* specifies the number of rows. *dynVal* indicates that the number of rows is specified dynamically.
Cols	*Cols* allows us to specify the number of columns of a matrix statically or to indicate that the number can be set at runtime (see Section 10.1.2.2.1.7).	
	Subfeatures	Possible values of *Rows*: *statVal*, *dynVal* *statVal* indicates that the number of columns is specified statically. The subfeature *ColsNumber* specifies the number of columns. *dynVal* indicates that the number of columns is specified dynamically.
RectFormat	*RectFormat* specifies the format of a rectangular matrix (see Section 10.1.2.2.1.3.6).	
	Subfeatures	Possible values of *RectFormat*: *array*, *CSC*, *CSR*, *COO* *array* implies column- or row-wise storage in a two-dimensional vector (see Section 10.1.2.2.1.3.6.1). *CSC* implies compressed sparse column format (see Section 10.1.2.2.1.3.6.2). *CSR* implies compressed sparse row format (see Section 10.1.2.2.1.3.6.2). COO coordinate format (see Section 10.1.2.2.1.3.6.2 and Table 41).

Table 31 *Description of subfeatures of rect*

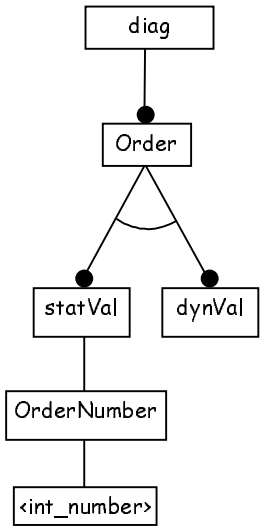

Figure 155 *Subfeatures of diag (see Table 32 for explanations)*

Feature name	Description	
Order	*Order* describes the number of columns and rows of a square matrix (see Section 10.1.2.2.1.7). We can specify order statically or indicate that it can be set at runtime.	
	Subfeatures	Possible values of *Order*: *statVal*, *dynVal* *statVal* indicates that order is specified statically. The subfeature *OrderNumber* specifies the order value. *dynVal* indicates that order is specified dynamically.

Table 32 *Description of Order, subfeature of diag, scalar, ident, zero, lowerTriang, upperTriang, symm, bandDiag, lowerBandDiag, and upperBandDiag*

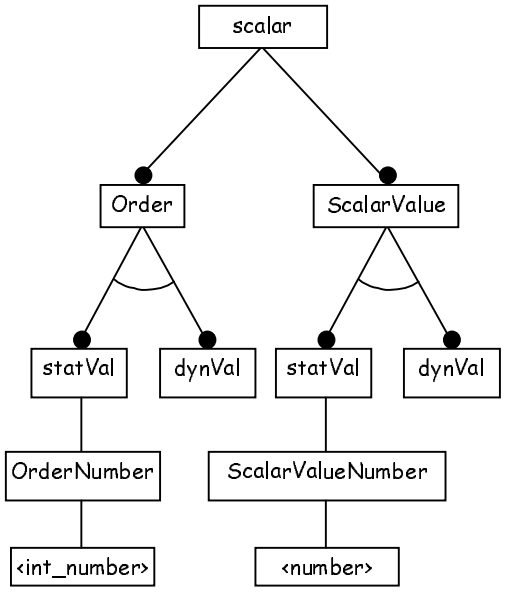

Figure 156 *Subfeatures of scalar (see Table 33 for explanations)*

Feature name	Description	
Order	see Table 32	
ScalarValue	*ScalarValue* allows us to specify the scalar value of a scalar matrix (i.e. the value of the diagonal elements) statically or to indicate that the value can be set at runtime (see Section 10.1.2.2.1.3.3).	
	Subfeatures	Possible values of *ScalarValue*: *statVal*, *dynVal* *statVal* indicates that the scalar value is specified statically. The subfeature *ScalarValueNumber* specifies the scalar value. *dynVal* indicates that the scalar value is specified dynamically.

Table 33 *Description of subfeatures of scalar*

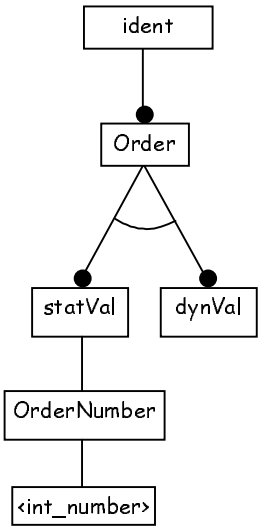

Figure 157 *Subfeatures of ident (see Table 32 for explanations)*

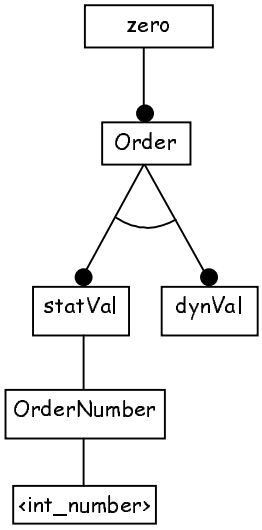

Figure 158 *Subfeatures of zero (see Table 32 for explanations)*

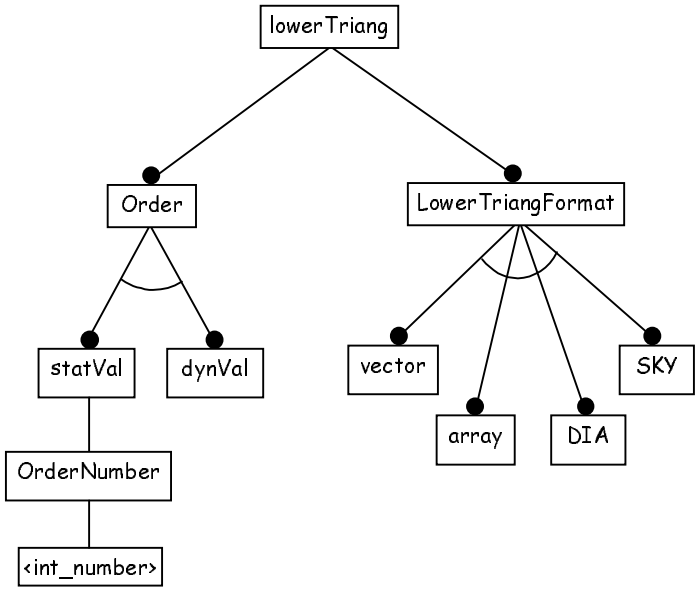

Figure 159 *Subfeatures of lowerTriang (see Table 34 for explanations)*

Feature name	**Description**	
Order	see Table 32	
LowerTriangFormat	*LowerTriangFormat* specifies the format of a lower triangular matrix (see Section 10.1.2.2.1.3.6).	
	Subfeatures	Possible values of *LowerTriangFormat*: *vector*, *array*, *DIA*, *SKY* *vector* implies diagonal-wise storage in a vector. *array* implies column- or row-wise storage in a two-dimensional vector (see Section 10.1.2.2.1.3.6.1). *DIA* implies diagonal sparse format (see Section 10.1.2.2.1.3.6.2). *SKY* implies skyline format (see Section 10.1.2.2.1.3.6.2).

Table 34 *Description of subfeatures of lowerTriang*

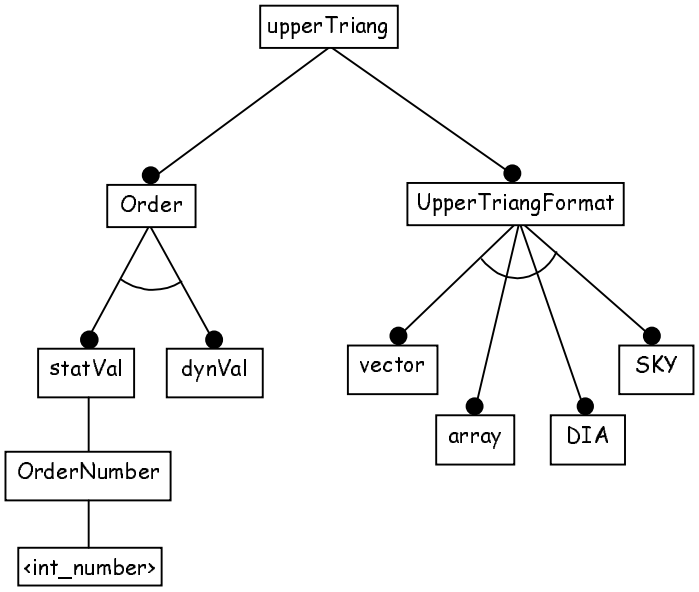

Figure 160 *Subfeatures of upperTriang (see Table 35 for explanations)*

Feature name	**Description**	
Order	see Table 32	
UpperTriangFormat	*UpperTriangFormat* specifies the format of an upper triangular matrix (see Section 10.1.2.2.1.3.6).	
	Subfeatures	Possible values of *UpperTriangFormat*: *vector*, *array*, *DIA*, *SKY* *vector* implies diagonal-wise storage in a vector. *array* implies column- or row-wise in a two-dimensional vector (see Section 10.1.2.2.1.3.6.1). *DIA* implies diagonal sparse format (see Section 10.1.2.2.1.3.6.2). *SKY* implies skyline format (see Section 10.1.2.2.1.3.6.2).

Table 35 *Description of subfeatures of upperTriang*

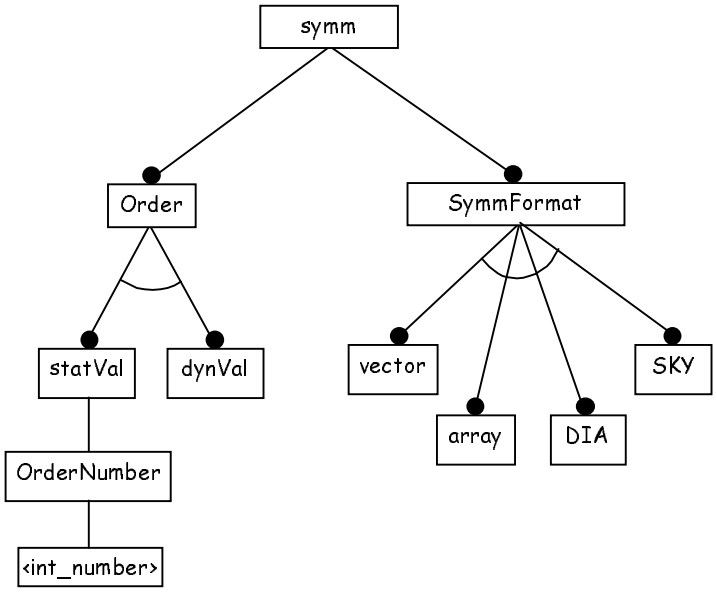

Figure 161 *Subfeatures of symm (see Table 36 for explanations)*

Feature name	Description	
Order	see Table 32	
SymmFormat	*SymmFormat* specifies the format of a symmetric matrix (see Section 10.1.2.2.1.3.6).	
	Subfeatures	Possible values of *SymmFormat*: *vector*, *array*, *DIA*, *SKY* *vector* implies diagonal-wise storage in a vector. *array* implies column- or row-wise storage in a two-dimensional vector (see Section 10.1.2.2.1.3.6.1). *DIA* implies diagonal sparse format (see Section 10.1.2.2.1.3.6.2). *SKY* implies skyline format (see Section 10.1.2.2.1.3.6.2).

Table 36 *Description of subfeatures of symm*

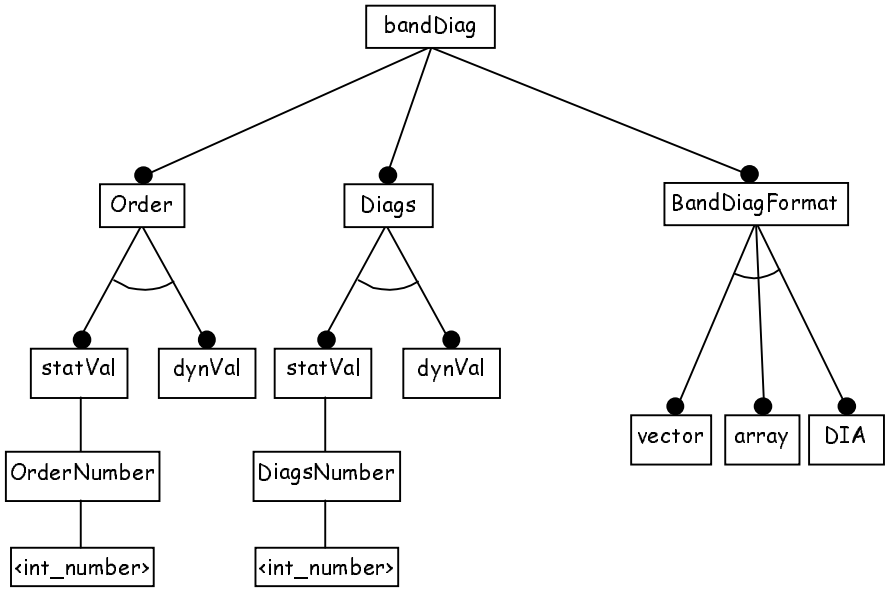

Figure 162 *Subfeatures of bandDiag (see Table 37 for explanations)*

Feature name	Description	
Order	see Table 32	
Diags	*Diags* allows us to statically specify the bandwidth (i.e. number of nonzero diagonals) of a band matrix or to indicate that the bandwidth can be set at runtime (see Section 10.1.2.2.1.7).	
	Subfeatures	Possible values of *Rows*: *statVal*, *dynVal* *statVal* indicates that the bandwidth is specified statically. The subfeature *DiagsNumber* specifies the number of nonzero diagonals. *dynVal* indicates that the bandwidth is specified dynamically.
BandDiagFormat	*BandDiagFormat* specifies the format of a band diagonal matrix (see Section 10.1.2.2.1.3.6).	
	Subfeatures	Possible values of *BandDiagFormat*: *vector*, *array*, *DIA* *vector* implies diagonal-wise storage in a vector. *array* implies column- or row-wise storage in a two-dimensional vector (see Section 10.1.2.2.1.3.6.1). *DIA* implies diagonal sparse format (see Section 10.1.2.2.1.3.6.2).

Table 37 *Description of subfeatures of bandDiag*

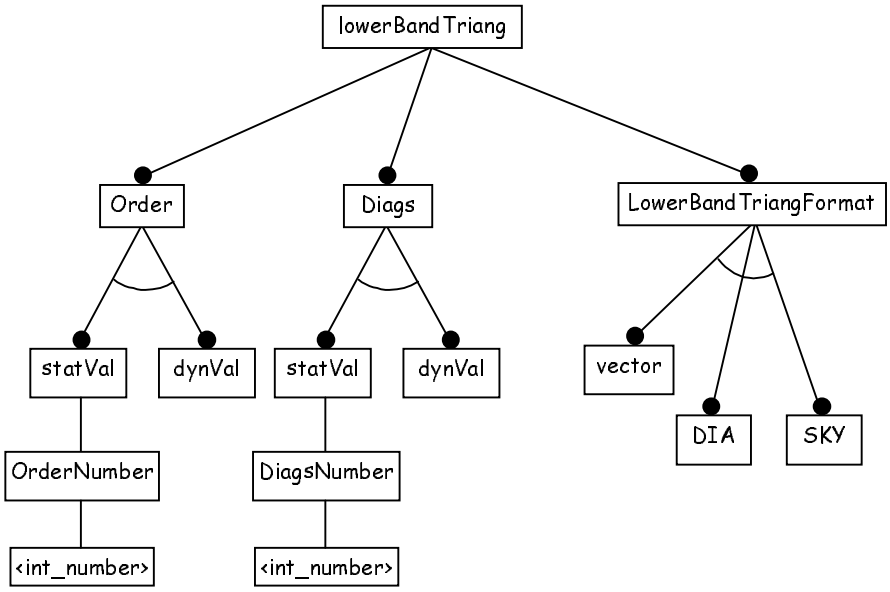

Figure 163 *Subfeatures of lowerBandTriang (see Table 38 for explanations)*

Feature name	Description	
Order	see Table 32	
Diags	see Diags in Table 37	
LowerBandTriangFormat	*LowerBandTriangFormat* specifies the format of a lower triangular matrix (see Section 10.1.2.2.1.3.6).	
	Subfeatures	Possible values of *LowerBandTriangFormat*: *vector*, *DIA*, *SKY* *vector* implies diagonal-wise storage in a vector. *DIA* implies diagonal sparse format (see Section 10.1.2.2.1.3.6.2). *SKY* implies skyline format (see Section 10.1.2.2.1.3.6.2).

Table 38 *Description of subfeatures of lowerBandTriang*

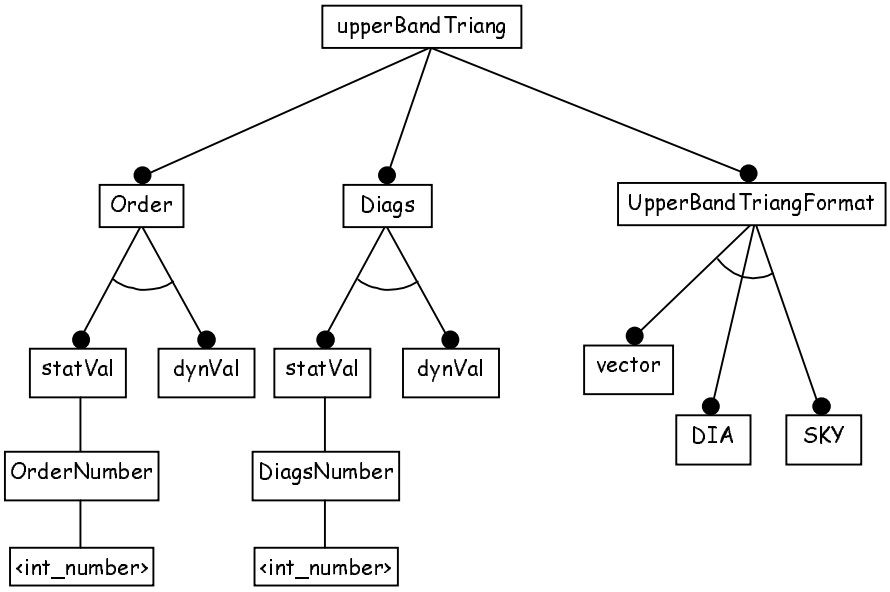

Figure 164 *Subfeatures of upperBandTriang (see Table 39 for explanations)*

Feature name	Description	
Order	see Table 32	
Diags	see Diags in Table 37	
UpperBandTriangFormat	*UpperBandTriangFormat* specifies the format of an upper band triangular matrix (see Section 10.1.2.2.1.3.6).	
	Subfeatures	Possible values of *UpperBandTriangFormat*: *vector*, *DIA*, *SKY* *vector* implies diagonal-wise storage in a vector. *DIA* implies diagonal sparse format (see Section 10.1.2.2.1.3.6.2). *SKY* implies skyline format (see Section 10.1.2.2.1.3.6.2).

Table 39 *Description of subfeatures upperBandTriang*

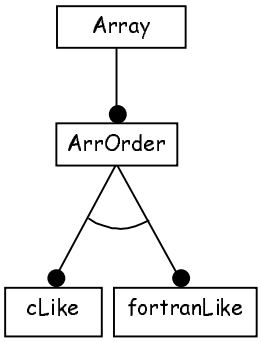

Figure 165 *Subfeatures of Array (see Table 40 for explanations)*

Feature name	Description	
ArrOrder	*ArrOrder* specifies whether to store elements row- or column-wise (see Section 10.1.2.2.1.3.6.1).	
	Subfeatures	Possible values of *Array*: *cLike*, *fortranLike* *CLike* implies row-wise storage. *fortranLike* implies column-wise storage.

Table 40 *Description of subfeatures of Array*

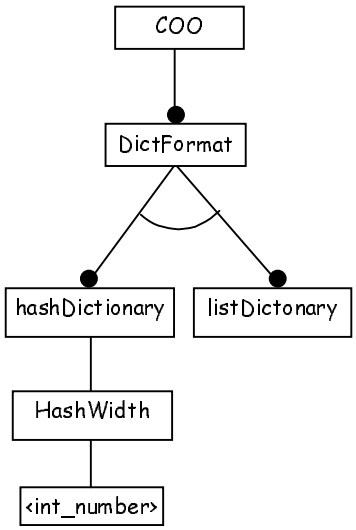

Figure 166 *Subfeatures of COO (see Table 41 for explanations)*

Feature name	Description	
DictFormat	*DictFormat* specifies the dictionary to be used for the COO matrix format (see Section 10.1.2.2.1.3.6.2).	
	Subfeatures	Possible values of *DictFormat*: *hashDict*, *listDict* *hashDict* implies a dictionary with a hashed key. The subfeature *HashWidth* is used in the hash function.[158] *listDict* implies a dictionary implemented using three vectors: two index vectors (for rows and columns indices) and one value vector.

Table 41 *Description of subfeatures of COO*

10.2.3.3 Direct Feature Defaults

A subset of the Matrix Configuration DSL parameters have direct defaults. In Table 42, we propose some, to our taste, reasonable default values. If necessary, they can be easily modified. Please note that some values are specified as ---. This symbol indicates that no reasonable default value exists for the corresponding parameter and, if the parameter is relevant in a given configuration, it has to be specified explicitly.

ElementType:	double
Structure:	structure
Shape:	rect
Malloc:	dyn
ArrOrder:	cLike
DictFormat:	hashDictionary
OptFlag:	space
ErrFlag:	checkAsDefault
IndexType:	unsigned int
Rows:	dynVal
Cols:	dynVal
Order:	dynVal
Diags:	dynVal
ScalarValue:	dynVal
RowsNumber:	---
ColsNumber:	---
OrderNumber:	---
DiagsNumber:	---
ScalarValueNumber:	---
Size:	int_number[int, 100]
Ratio:	float_number[double, 0.1]
Growing:	float_number[double, 0.25]
HashWidth:	int_number[IndexType, 1013]

Table 42 *Direct feature defaults for the Matrix Configuration DSL [Neu98]*

Density represents a special case. If Density is specified and Format[159] is not, Density is used to compute Format. On the other hand, if Format is specified and Density is not, Format is used to compute Density. Finally, if neither Format nor Density is specified, we use the following default for Density (and then compute Format):

Density:	dense

10.2.3.4 Computed Feature Defaults

The default values of the following parameters are computed:

Density
MallocChecking
BoundsChecking
CompatChecking
RectFormat
LowerTriangFormat
UpperTriangFormat
SymmFormat
BandDiagFormat
LowerBandTriangFormat
UpperBandTriang

Table 43 through Table 51 specify how to compute them. The semantics of the feature dependencies tables are explained in Section 10.2.3.

10.2.3.4.1 Density

Format	Density
array	dense
vector	dense
CSR	sparse
CSC	sparse
COO	sparse
DIA	sparse
SKY	sparse
*	dense (from Section 10.2.3.3)

Table 43 *Computing default value for Density [Neu98]*

10.2.3.4.2 Error Checking Features

ErrFlag is used as the primary default for all error checking features (see Section 10.2.3).

ErrFlag	MallocChecking
checkAsDefault	checkMallocErr
noChecking	noMallocErrChecking

Table 44 *Computing default value for MallocChecking [Neu98]*

ErrFlag	BoundsChecking
checkAsDefault	checkBounds
noChecking	noBoundsChecking

Table 45 *Computing default value for BoundsChecking [Neu98]*

ErrFlag	CompatChecking
checkAsDefault	checkCompat
noChecking	noCompatChecking

Table 46 *Computing default value for CompatChecking [Neu98]*

10.2.3.4.3 Format Features

The default format for a certain matrix shape is determined based on Density and OptFlag. We already explained the relationship between shape, format and OptFlag in Section 10.2.3

OptFlag	Density	RectFormat
*	dense	array
*	sparse	COO

Table 47 *Computing default value for RectFormat [Neu98]*

OptFlag	Density	LowerTriangFormat or UpperTriangFormat
speed	dense	array
speed	sparse	DIA
space	dense	vector
space	sparse	SKY

Table 48 *Computing default value for LowerTriangFormat and UpperTriangFormat [Neu98]*

OptFlag	Density	SymmFormat
speed	dense	array
space	dense	vector
*	sparse	SKY

Table 49 *Computing default value for SymmFormat [Neu98]*

OptFlag	Density	BandDiagFormat
*	dense	array
*	sparse	DIA

Table 50 *Computing default value for BandDiagFormat [Neu98]*

OptFlag	Density	LowerBandTriangFormat or UpperBandTriangFormat
*	dense	vector
speed	sparse	DIA
space	sparse	SKY

Table 51 *Computing default value for LowerBandTriangFormat and UpperBandTriang [Neu98]*

10.2.3.5 Flat Configuration Description

A concrete matrix type is completely described by specifying the values of the DSL parameters for which two or more alternative values are possible. We refer to the set of these DSL parameters as the *"flat" configuration description* (or simply *"flat" configuration*) or the *type record* of a matrix. The flat configuration for the matrix component is the following set of DSL parameters:

ElementType
Shape
Format
ArrOrder
DictFormat
Density
Malloc
MallocErrChecking
OptFlag
ErrFlag
BoundsChecking
CompatChecking
IndexType
Rows
Cols
Order
Diags
ScalarValue
Ratio
Growing
RowsNumber
ColsNumber
OrderNumber
DiagsNumber
Size
HashWidth
ScalarValueNumber

Please note that we combined all format parameters into Format since only one of them is used at a time. Furthermore, not all parameters are relevant in all cases. For example, if Format is vector, DictFormat is irrelevant.

The flat configuration is computed by analyzing the given matrix configuration expression and assigning defaults to the unspecified features.

10.2.4 Matrix ICCL

We construct a concrete matrix type by composing a number of *matrix implementation components*. The matrix package contains a set of parameterized implementation components, some of which are alternative and some optional. The parameterized components can be configured in a number of different ways. The exact specification of valid configurations is given by a grammar. This grammar specifies the *Matrix Implementations Components Configuration Language*, i.e. Matrix ICCL. In our case, the ICCL is specified by a GenVoca-style grammar, which we will show in Section 10.2.4.8. But before that, we first give an overview of the available matrix implementation components.

The matrix implementation components are organized into a number of generic layers (see Figure 167; we discussed this kind of diagrams in Section 6.4.2.2). The layers provide a high-level overview of the relationships between the components and how they are used to construct concrete matrix types.

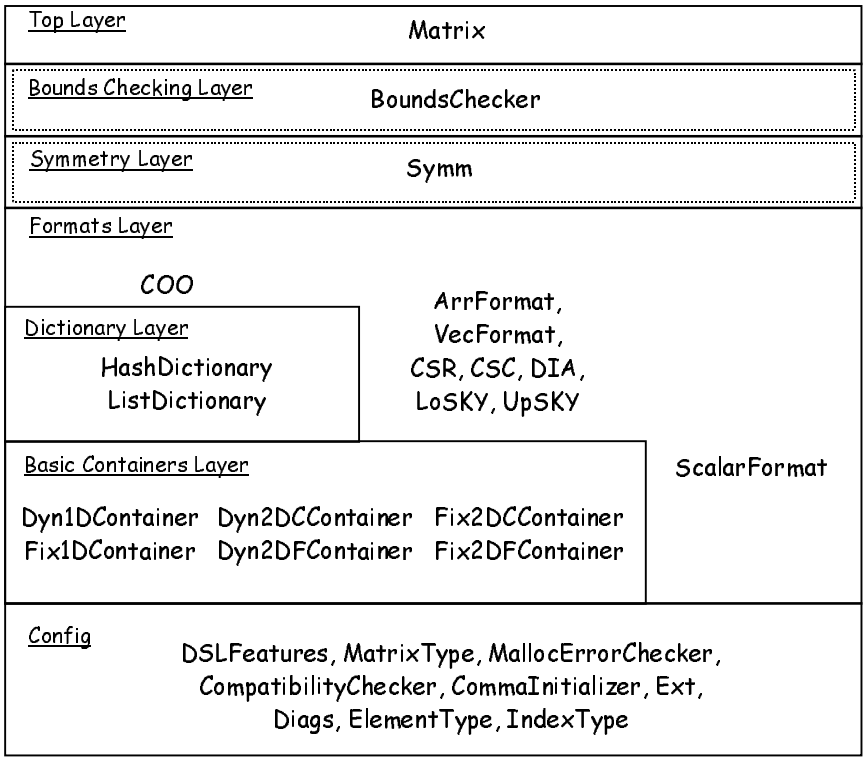

Figure 167 *Layered matrix implementation component*

We have the following layers (the names of the layers are underlined in Figure 167):

- *Top layer*: The top layer consists of only one component, which is Matrix. Matrix represents the outer wrapper of every matrix configuration. Its purpose is to express their commonality as matrices. This is useful for writing generic operations on matrices.

- *Bounds checking layer*: This layer contains the optional wrapper BoundsChecker, which provides bounds checking functionality.

- *Symmetry layer*: This layer contains the optional wrapper Symm, which implements symmetry. Symm turns a lower triangular format into a symmetric format.

- *Formats layer*: Formats layer provides components implementing various dense and sparse storage formats.

- *Dictionary layer*: Dictionaries are required by the COO format.

- *Basic containers layer*: Any format that physically stores matrix elements uses basic containers.

The box at the bottom of Figure 167 is the *configuration repository*, which we also refer to as Config. Any component from any layer can reach down to the repository and get the information it needs (we already explained this idea in Sections 6.4.2.4 and 8.7). In particular, the repository

is used to store some global components, the type of the matrix being constructed, some horizontal parameters (see Section 6.4.2.2), all the abstract matrix features which were computed from the matrix configuration description (i.e. the flat configuration, which is contained in DSLFeatures) and the description itself.

Each of the matrix components is parameterized. The basic containers are parameterized with Config. The dictionaries are parameterized with the basic containers. The formats are parameterized either with Config, or with the basic containers, or with the dictionaries. More precisely, ScalarFormat is parameterized with Config, whereas COO is parameterized with the dictionaries and the remaining components in this layer are parameterized with the basic containers. Symm is parameterized with the formats. Since the symmetry layer has a dashed inner box, Symm is an optional component. Thus, BoundsChecker can be parameterized with Symm or with the formats directly. Similarly, Matrix can be parameterized with BoundsChecker, or with Symm, or with the formats.

The names appearing in the Config box are variables rather than components. They are assigned concrete components when a configuration is built, so that other components can access them as needed.

An example of a concrete matrix configuration is shown in Figure 168. As we will see later, some formats and the dictionaries use more than one different basic containers at once. Thus, in general, a matrix configuration can look like a tree rather than a stack of components.

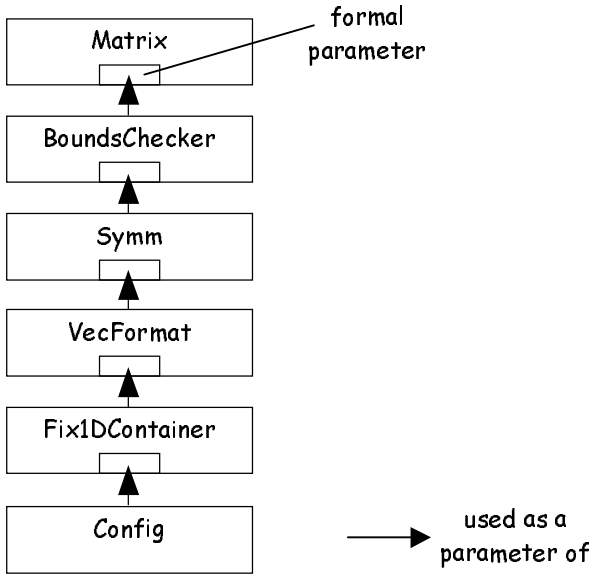

Figure 168 *Example of a matrix configuration*

Now, let us take a closer look at each group of matrix implementation components starting with the bottom layer.

10.2.4.1 Basic Containers

Basic containers are the basic components for storing objects. They are used by the matrix format components to store matrix elements and, in some cases, also element indices and pointers to containers. They are also used to implement dictionaries.

We have six basic containers (see Table 52). They can all be thought of as one or two-dimensional arrays. The containers whose names start with "Dyn" allocate their memory dynamically and the

ones starting with "Fix" use static memory allocation (i.e. the size of the allocated memory is specified at compile time). The next two characters in the name of a container indicate whether the container is one or two dimensional. Finally, the two-dimensional containers store their elements either row-wise (this is indicated with an extra C, which stands for C-like storage) or column-wise (which is indicated by an extra F, which stands for Fortran-like storage).

Basic Container	Memory Allocation	Number of Dimensions	Storage Format
Dyn1Dcontainer	dynamic	1	---
Fix1Dcontainer	static	1	---
Dyn2DCContainer	dynamic	2	row-wise
Dyn2DFContainer	dynamic	2	column-wise
Fix2DCContainer	static	2	row-wise
Fix2DFContainer	static	2	column-wise

Table 52 *Basic containers (adapted from [Neu98])*

The basic containers have the following parameters:

```
Dyn1DContainer[ElementType, Ratio, Growing, Config]
Fix1DContainer[ElementType, Size, Config]
Dyn2DCContainer[Config]
Dyn2DFContainer[Config]
Fix2DCContainer[Size, Config]
Fix2DFContainer[Size, Config]
```

ElementType is the type of the elements stored in the container. Only the one-dimensional containers have ElementType as their explicit parameter since they are sometimes used more than once in a configuration, e.g. to store matrix elements and matrix element indices. The two-dimensional containers, on the other hand, are only used to store matrix elements. Thus, they can get their element type from Config (which is the element type of the whole matrix). We already explained Size, Ratio, Growing in Table 29. All containers get their index types from Config.

In general, we do not really have to specify ElementType as an explicit parameter of the one-dimensional containers. This is so since the component which gets a container as its parameter can internally request the vector storage type from the container and specify the element type at this time. For example, we could pass just one container to the CSR-format component, and the component would internally request two different types from this container: one vector of index type and one vector of element type. We illustrated this idea with some C++ code in Figure 169.

The solution in Figure 169 has the advantage that we do not have to specify ElementType as an explicit parameter of Dyn1Dcontainer. This leads to a much simpler description of the possible component combinations: We would be able to replace the three productions VerticalContainer, IndexVec, and ElemVec later in Figure 173 with just one production:

```
Vec:        Dyn1DContainer[Ratio, Growing, Config] | Fix1DContainer[Size, Config]
```

Unfortunately, we were not able to use this variant since VC++5.0 reported an internal error when compiling this valid C++ code.

```
//int_number and Configuration are not shown here
...

template<class Size, class Config>
class Dyn1DContainer
{
   public:
      //export Config
      typedef Config Config;

      //components using me can request a Vector as many times as they want;
      //they can specify a different ElementType each time
      template<class ElementType>
      class Vector
      {
         ...
      };
};

template<class Container1D>
class CSR
{
   public:
      //retrieve matrix index type and matrix element type from Config
      typedef Container1D::Config Config;
      typedef Config::IndexType IndexType;
      typedef Config::ElementType ElementType;

      //the CSR component requests the index vector and the element vector
      Container1D::Vector< IndexType> indexVec;
      Container1D::Vector< ElementType> elementVec;
      ...
};

main()
{
  CSR<Dyn1DContainer<int_number<long, 100>, Configuration> > myFormat;
}
```

Figure 169 *C++ Code demonstrating how CSR could request two vectors with different element types from a 1D container (adapted from [Neu98])*

10.2.4.2 Dictionaries

Dictionaries are used by the coordinate matrix format, i.e. COO. Their elements are addressed by a pair of integral numbers. Many variant implementations of a dictionary are possible. However, we consider here only two variants:

```
HashDictionary[VerticalContainer, HorizontalContainer, HashFunction]
ListDictionary[IndexVector, ElementVector]
```

HashDictionary uses a hashing function to index VerticalContainer, which contains references to element buckets (see Figure 170). We substitute one of the one-dimensional basic containers with a constant size for VerticalContainer. The buckets themselves are instantiated from HorizontalContainer. HorizontalContainer is also a dictionary and we use ListDictionary here. HashFunction is a component providing the hash function.

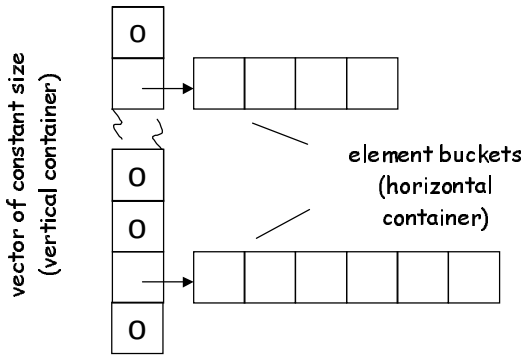

Figure 170 *Structure of a hash dictionary (from [Neu98])*

ListDictionary is a dictionary implementation storing element indices in two instances of IndexVector and elements in ElementVector. Both vector parameters are substituted by one of the one-dimensional basic containers.

10.2.4.3 Formats

We have nine matrix format components. They are summarized in Table 53.

Format Component	Purpose
ArrFormat	ArrFormat stores elements in a two-dimensional basic container. If the indices of a matrix element are i and j, the same indices are used to store the element in the container. ArrFormat is used to store dense matrices of different band shapes (see Figure 141). It optionally checks to make sure that nonzero elements are assigned to locations within the specified band only.
VecFormat	VecFormat stores elements in a one-dimensional basic container (thus, it uses a formula to convert matrix element indices into the container element indices). The elements are stored diagonal-wise. It is used to store dense matrices of different band shapes. Only the nonzero diagonals are stored. It optionally checks to make sure that nonzero elements are assigned to locations within the specified band only.
ScalarFormat	ScalarFormat is used to represent scalar matrices (including zero and identity matrix). The scalar value can be specified statically or dynamically. This format does not require any storage container.
CSR	CSR implements the compressed sparse row format. CSR is used for general rectangular sparse matrices.
CSC	CSC implements the compressed sparse column format. CSC is used for general rectangular sparse matrices.
COO	COO implements the coordinate format. COO is used for general sparse rectangular matrices.
DIA	DIA implements the sparse diagonal format. DIA is used for sparse band matrices
LoSKY	LoSKY implements the lower skyline format. LoSKY is used for sparse lower triangular matrices and sparse lower band triangular matrices.
UpSKY	UpSKY implements the upper skyline format. UpSKY is used for sparse upper triangular matrices and sparse upper band triangular matrices.

Table 53 *Format components (see Section 10.1.2.2.1.3.6.2 for the explanation of the sparse formats)*

The parameters of the format components are as follows:

ArrFormat[Ext, Diags, Array]

VecFormat[Ext, Diags, ElemVec]
ScalarFormat[Ext, ScalarValue, Config]
CSR[Ext, IndexVec, ElemVec]
CSC[Ext, IndexVec, ElemVec]
COO[Ext, Dict]
DIA[Ext, Diags, Array]
LoSKY[Ext, Diags, IndexVec, ElemVec]
UpSKY[Ext, Diags, IndexVec, ElemVec]

Array is any of the two-dimensional basic containers. ElemVec and IndexVec are any of the one-dimensional basic containers. Ext and Diags need a bit more of explanation.

10.2.4.3.1 Extent and Diagonal Range of a Matrix

The *extent* of a matrix is determined by the number of rows and the number of columns. The *diagonal range* specifies the valid diagonal index range for band matrices. The index of a diagonal is explained in Figure 171. For example, the main diagonal has the index 0. The range -1...1 means that only the diagonals -1, 0, and 1 can contain nonzero elements.

$$\begin{pmatrix} 0 & 1 & & & n-1 \\ -1 & 0 & 1 & \iddots & \\ & -1 & 0 & 1 & \\ & & -1 & 0 & 1 \\ & & & -1 & 0 \\ & \iddots & & & -1 \\ 1-m & & & & \end{pmatrix}$$

Figure 171 *Numbering of the diagonals of a m-by-n matrix (adapted from [Neu98])*

Table 54 summarizes the extent and diagonal range of some more common matrix shapes.

Shape	Extent	Diagonal range
rectangular	arbitrary m and n	-m...n
diagonal	m = n	0...0
lower triangular	m = n	-m...0
upper triangular	m = n	0...n
symmetric	m = n	-m...n (or –m...0)[160]
band diagonal	m = n	-⌊d/2⌋...⌊d/2⌋, where ⌊⌋ denote rounding down
lower band triangular	m = n	1-d...0
upper band triangular	m = n	0...d-1

Table 54 *Extent and diagonal range of some more common matrix shapes (from [Neu98]). d is the number of diagonals.*

Ext

The parameter Ext is used to specify the extent of a matrix, i.e. the number of rows and the number of columns and whether they are determined statically or dynamically. We have a number of small helper components that can be used in place of Ext to specify the extent of rectangular and square matrices. They are described in Figure 172.

Extent Component	Extent	Number of rows / columns
DynExt	rectangular	both dynamic
StatExt	rectangular	both static
DynRowsStatCols	rectangular	row number dynamic, column number static
StatRowsDynCols	rectangular	row number static, column number dynamic
DynSquare	square	both dynamic
StatSquare	square	both static

Figure 172 *Helper components for specifying the extent of a matrix (adapted from [Neu98])*

The extent components have the following parameters:

```
DynExt[IndexType]
StatExt[Rows, Cols]
StatRowsDynCols[Rows]
DynRowsStatCols[Cols]
DynSquare[IndexType]
StatSquare[Rows]
```

Rows specifies the number of rows and Cols the number of columns. IndexType specifies the index type. Rows and Cols can be specified as follows:

```
int_number[Type, value]
```

Diags

The parameter Diags specifies the diagonal range of a matrix format. The helper components which can be used in place of Diags are described in Table 55.

Diags Component	Purpose
DynLo	DynLo specifies lower band triangular shape with dynamic bandwidth. The diagonal range is 1-d...0, where d is specified dynamically.
DynUp	DynUp specifies upper band triangular shape with dynamic bandwidth. The diagonal range is 0...d-1, where d is specified dynamically.
DynBand	DynBand specifies band diagonal shape with dynamic bandwidth. The diagonal range is -⌊d/2⌋...⌊d/2⌋, where d is specified dynamically.
TriangLo	TriangLo specifies lower triangular shape.
TriangUp	TriangUp specifies upper triangular shape.
Rect	Rect specifies rectangular shape.
StatDiags[FirstDiag, LastDiag]	StatDiags allows to statically specify the band range. The diagonal range is FirstDiag...LastDiag.
StatBand[BandWidth]	StatBand specifies band diagonal shape with static bandwidth. The diagonal range is -⌊BandWidth /2⌋... ⌊BandWidth /2⌋.

Table 55 *Helper components for specifying components the diagonal range of a matrix*

The Diags components have the following parameters:

```
DynLo[IndexType]
DynUp[IndexType]
DynBand[IndexType]
TriangLo[IndexType]
```

TriangUp[IndexType]
Rect[IndexType]
StatDiags[FirstDiag, LastDiag]

10.2.4.4 Symmetry Wrapper

As described in Table 20, the elements of a symmetric matrix can be stored in a lower (or upper) triangular matrix. Thus, we do not need a symmetric format storage component. Instead, we implement the symmetry property using a wrapper which we can put on some other format component. The wrapper maps any element access to it onto a corresponding element access to the lower (or upper) half of the wrapped format component. The wrapper takes a format component as its parameter:

Symm[Format]

The symmetry wrapper can be used with any of the available format components. However, only the lower part of the component is actually used and thus only some combinations are relevant. For example, we would use a lower triangular format to store a symmetric matrix and lower band triangular format to store a symmetric band diagonal matrix.[161] Putting the symmetry wrapper on a diagonal shape is not particularly useful. Please remember that the shape of a format component depends on the component, the extent, and the diagonal range.

10.2.4.5 Error Checking Components

Currently, we provide components for bounds checking (see Section 10.1.2.2.1.4.1), compatibility checking (see Section 10.1.2.2.1.4.2), and memory allocation error checking (the latter are used by the dynamic basic containers).

Bounds checking is best implemented in the form of a wrapper since it involves precondition checking and thus we only need to wrap the element access methods of the underlying format component. Thus, BoundsChecker takes format as its only parameter:

BoundsChecker[FormatOrSymm]

The compatibility checking and memory allocation error checking components, on the other hand, are called from within methods of other components, i.e. they involve intracondition checking. Thus, we do not implement them as wrappers. They are implemented as stand alone components and we provide them to the configuration through the MallocErrorChecker and CompatibilityChecker variables of the configuration repository (i.e. Config). Any component that needs them can retrieve them from the repository.

The compatibility checking and memory allocation error checking components are described in Table 56 and Table 57. They all take Config as their parameter.

CompatibilityChecker Component	Purpose
CompatChecker	CompatChecker is used to check the compatibility of matrices for addition, subtraction, and multiplication. It is called from the code implementing the matrix operations.
EmptyCompatChecker	EmptyCompatChecker implements its error checking methods as empty methods (please note that inlining will eliminate any overhead). If used, no compatibility checking is done.

Table 56 *Compatibility checking components*

MallocErrorChecker Component	Purpose
MallocErrChecker	MallocErrChecker is used to check for memory allocation errors in the dynamic basic containers.
EmptyMallocErrChecker	EmptyMallocErrChecker implements its error checking methods as empty methods. If used, no memory allocation error checking is done.

Table 57 *Memory allocation error checking component*

10.2.4.6 Top-Level Wrapper

Each matrix configuration is wrapped into Matrix. This wrapper is used to express the type commonality of all matrix configurations. This is particularly useful for generic operations so that they can check whether the type they operate on is a matrix type or not.

10.2.4.7 Comma Initializers

Comma initializers implement matrix initialization using comma separated lists of numbers (as implemented in the Blitz++ library [Vel97]). These components are specific to the C++ implementation.

Using a comma initializer, we can initialize a dense matrix by listing its elements (the matrix already knows its extent), e.g.:

```
matrix= 3, 0, 0, 8, 7,
        0, 2, 1, 2, 4,
        6, 0, 2, 4, 5;
```

For sparse matrices, the initialization format is different. We specify the value of an element followed by its indices, e.g.:

```
sparseMatrix= 3, 0, 0,
              1, 1, 2,
              2, 1, 1,
              6, 2, 0,
              2, 2, 2;
```

In general, a dense m-by-n matrix is initialized as follows [Neu98]:

$$\begin{aligned} A= \quad & a_{11}, \quad a_{12}, \quad \cdots \quad a_{1n}, \\ & a_{21}, \quad \ddots \qquad\quad a_{2n}, \\ & \vdots \qquad\quad \ddots \quad \vdots \\ & a_{m1}, \quad \ldots \quad \ldots \quad a_{mn}; \end{aligned}$$

and a sparse m-by-n matrix:

$$\begin{aligned} A= \quad & a_{i_1 j_1}, \quad i_1, \quad j_1, \\ & a_{i_2 j_2}, \quad i_2, \quad j_2, \\ & \vdots \\ & a_{i_k j_k}, \quad i_k, \quad j_k; \end{aligned}$$

The available comma initialization components are summarized in Table 58.

CommaInitializer Component	Purpose
DenseCCommaInitializer	DenseCCommaInitializer is used for initializing dense matrices with row-wise storage.
DenseFCommaInitializer	DenseFCommaInitializer is used for initializing dense matrices with column-wise storage.
SparseCommaInitializer	SparseCommaInitializer is used for initializing sparse matrices.

Table 58 *Comma initialization components*

The comma initializer currently used by a configuration is published in the configuration repository in the CommaInitializer component variable.

10.2.4.8 Matrix ICCL Grammar

The complete ICCL grammar is specified in the GenVoca-like notation in Figure 173.

MatrixType:	Matrix[OptBoundsCheckedMatrix]
OptBoundsCheckedMatrix:	OptSymmetricMatrix | BoundsChecker[OptSymmetricMatrix]
OptSymmetricMatrix:	Format | Symm[Format]
Format:	ArrFormat[Ext, Diags, Array] | VecFormat[Ext, Diags, ElemVec] | ScalarFormat[Ext, ScalarValue, Config] | CSR[Ext, IndexVec, ElemVec] | CSC[Ext, IndexVec, ElemVec] | COO[Ext, Dict] | DIA[Ext, Diags, Array] | LoSKY[Ext, Diags, IndexVec, ElemVec] | UpSKY[Ext, Diags, IndexVec, ElemVec]
Dict:	HashDictionary[VerticalContainer, HorizontalContainer, HashFunction] | ListDictionary[IndexVec, ElemVec]
Array:	Dyn2DCContainer[Config] | Fix2DCContainer[Size, Config] | Dyn2DFContainer[Config] | Fix2DFContainer[Size, Config]
HorizontalContainer:	ListDictionary[IndexVector, ElementVector]
VerticalContainer:	Dyn1DContainer[HorizPointer, Ratio, Growing, Config] | Fix1DContainer[HorizPointer, Size, Config]
IndexVec:	Dyn1DContainer[IndexType, Ratio, Growing, Config] | Fix1DContainer[IndexType, Size, Config]
ElemVec:	Dyn1DContainer[ElementType, Ratio, Growing, Config] | Fix1DContainer[ElementType, Size, Config]
HashFunction:	SimpleHashFunction[HashWidth]
Ext:	DynExt[IndexType] | StatExt[Rows, Cols] | DynSquare[IndexType] | StatSquare[Rows] | StatRowsDynCols[Rows] | DynRowsStatCols[Cols]
Diags:	DynLo[IndexType] | DynUp[IndexType] | DynBand[IndexType] | TriangLo[IndexType] | TriangUp[IndexType] | Rect[IndexType] | StatDiags[FirstDiag, LastDiag] | StatBand[BandWidth]
ScalarValue:	DynVal[ElementType] | StatVal[Val]
MallocErrorChecker:	EmptyMallocErrChecker[Config] | MallocErrChecker[Config]
CompatibilityChecker:	EmptyCompatChecker[Config] | CompatChecker[Config]
CommaInitializer:	DenseCCommaInitializer[Config] | DenseFCommaInitializer[Config] | SparseCommaInitializer[Config]
Ratio, Growing:	float_number[Type, value]
Rows, Cols, FirstDiag, LastDiag, BandWidth, Size, HashWidth:	int_number[Type, value]
Val:	int_number[Type, value] | float_number[Type, value]
ElementType:	float | double | long double | short | int | long | unsigned short | unsigned int | unsigned long
IndexType:	char | short | int | long | unsigned char | unsigned short | unsigned int | unsigned long | signed char
SignedIndexType:	char | short | int | long
HorizPointer:	HorizontalContainer*
Config:	Configuration (contains the component variables: DSLFeatures, MatrixType, MallocErrorChecker, CompatibilityChecker, CommaInitializer, ElementType, IndexType, Ext, Diags)
DSLFeatures:	this is the "flat" configuration

Figure 173 *Matrix ICCL [Neu98]*

Here is an example of a valid ICCL expression:

```
Matrix[
  BoundsChecker[
    ArrFormat[
      DynExt[unsigned int],
      Rect[unsigned int],
      Dyn2DCContainer[Config]
    ]
  ]
]
```

10.2.5 Mapping from Matrix Configuration DSL to Matrix ICCL

Now that we completely specified the Matrix Configuration DSL and the Matrix ICCL, we need to specify how to translate a Matrix Configuration DSL expression into a Matrix ICCL expression. We will use the dependency tables introduced in Section 10.2.3 for this purpose. Each table will specify how to compute an ICCL parameter based on some DSL parameters from the flat configuration description (Section 10.2.3.5).

10.2.5.1 Basic Containers

We start with the basic containers. The mapping for Array is shown in Table 59. The value of Array depends on the DSL features Malloc and ArrOrder.

Malloc	ArrOrder	Array
fix	cLike	Fix2DCContainer
	fortranLike	Fix2DFContainer
dyn	cLike	Dyn2DCContainer
	fortranLike	Dyn2DFContainer

Table 59 *Table for computing the ICCL parameter Array*

Table 60 specifies how to compute IndexVec, ElemVec, and VerticalContainer.

Malloc	IndexVec, ElemVec, VerticalContainer
fix	Fix1Dcontainer
dyn	Dyn1Dcontainer

Table 60 *Table for computing the ICCL parameters IndexVec, ElemVec, and VerticalContainer [Neu98]*

The following parameters of the basic containers are determined directly from the DSL parameters:

```
Size (ICCL)     = Size (DSL)
Ratio (ICCL)    = Ratio (DSL)
Growing (ICCL)  = Growing (DSL)
ElementType (ICCL)  = ElementType (DSL)
IndexType (ICCL)    = IndexType (DSL)
```

Since some DSL parameters have the same name as the corresponding ICCL parameters, we indicate whether we refer to a DSL parameter or to a ICCL parameter by an extra annotation.

The signed index type (SignedIndexType) is used for the diagonal indices in the DIA format component. SignedIndexType is determined based on IndexType. This is specified in Table 61. Please note that this mapping may be problematic for very large matrices with unsigned index type and using DIA since only the have of the row (or column) index range is available for indexing the diagonals.

IndexType	SignedIndexType
unsigned char	signed char
unsigned short	short
unsigned int	int
unsigned long	long
*	=IndexType

Table 61 *Table for computing the ICCL parameter SignedIndexType [Neu98]*

10.2.5.2 Dictionaries

A dictionary component is selected based on the DSL feature DictFormat (see Table 62).

DictFormat	Dict
hashDictionary	HashDictionary
listDictionary	ListDictionary

Table 62 *Table for computing the ICCL parameter Dict [Neu98]*

Currently, two of the parameters of HashDictionary use direct defaults:

```
HorizontalContainer   = ListDictionary
HashFunction    = SimpleHashFunction[HashWidth]
```

Additionally, we have the following dependencies:

```
HashWidth (ICCL)    = HashWidth (DSL)
HorizPointer    = HorizontalContainer*    (C++ notation for pointers)
```

10.2.5.3 Formats

The ICCL parameter Format is determined based on DSL features Shape and Format. The mapping is given in Table 63. Please note that the DSL feature Density was already used for computing the flat configuration DSL parameter Format (see Section 10.2.3.4.3).

Shape	Format (DSL)	Format (ICCL)
diag	---	VecFormat
scalar	---	ScalarFormat
ident	---	ScalarFormat
zero	---	ScalarFormat
*	array	ArrFormat
	vector	VecFormat
	CSR	CSR
	CSC	CSC
	COO	COO
	DIA	DIA
lowerTriang symm lowerBandTriang	SKY	LoSKY
upperTriang upperBandTriang	SKY	UpSKY

Table 63 *Table for computing the ICCL parameter Format [Neu98]*

The remaining tables specify the mapping for the various parameters of the format components.

Shape			Ext
	Rows (DSL)	Cols (DSL)	
rect	dynVal	dynVal	DynExt
	dynVal	statVal	StatRowsDynCols
	statVal	dynVal	DynRowsStatCols
	statVal	statVal	StatExt
	Order		
*	dynVal		DynSquare
	statVal		StatSquare

Table 64 *Table for computing the ICCL parameter Ext [Neu98]*

Shape	Diags (DSL)	Diags (ICCL)
rect	---	Rect
diag	---	StatBand
scalar	---	StatBand
ident	---	StatBand
zero	---	StatBand
lowerTriang	---	TriangLo
upperTriang	---	TriangUp
symm	---	TriangLo
bandDiag	dynVal	DynBand
	statVal	StatBand
lowerBandTriang	dynVal	DynLo
	statVal	StatDiags
upperBandTriang	dynVal	DynUp
	statVal	StatDiags

Table 65 *Table for computing the ICCL parameter Diags [Neu98]*

Shape	Diags (DSL)	FirstDiag	LastDiag
lowerBandTriang	statVal	=1-DiagsNumber	number[IndexType,0]
upperBandTriang	statVal	number[IndexType,0]	=DiagsNumber-1

Table 66 *Table for computing the ICCL parameters FirstDiag and LastDiag [Neu98]*

Shape	Diags (DSL)	BandWidth
diag scalar ident zero	---	number[IndexType,1]
bandDiag	statVal	DiagsNumber

Table 67 *Table for computing the ICCL parameter BandWidth [Neu98]*

ScalarValue (DSL)	ScalarValue (ICCL)
dynVal	DynVal
statVal	StatVal

Table 68 *Table for computing the ICCL parameter ScalarValue [Neu98]*

The ICCL parameter Val is computed as follows:

Val =ScalarValueNumber

Shape	Rows (ICCL)	Cols (ICCL)
rect	= RowsNumber	= ColsNumber
*	= OrderNumber	= OrderNumber

Table 69 *Table for computing the ICCL parameters Rows and Cols [Neu98]*

Finally, Shape determines whether we use the Symm wrapper or not.

Shape	OptSymmetricMatrix
symm	Symm
*	= Format (ICCL)

Table 70 *Table for computing the ICCL parameter OptSymmetricMatrix [Neu98]*

10.2.5.4 Error Checking Components

BoundsChecking	OptBoundsCheckedMatrix
checkBounds	BoundsChecker
noBoundsChecking	= OptSymmetricMatrix

Table 71 *Table for computing the ICCL parameter OptBoundsCheckedMatrix [Neu98]*

MallocErrChecking	MallocErrorChecker
checkMallocErr	MallocErrChecker
noMallocErrChecking	EmptyMallocErrChecker

Table 72 *Table for computing the ICCL parameter MallocErrorChecker [Neu98]*

Shape	CompatibilityChecker
checkCompat	CompatChecker
noCompatChecking	EmptyCompatChecker

Table 73 *Table for computing the ICCL parameter CompatibilityChecker [Neu98]*

10.2.5.5 Comma Initializer

Density	ArrOrder	CommaInitializer
dense	cLike	DenseCCommaInitializer
	fortranLike	DenseFCommaInitializer
sparse	---	SparseCommaInitializer

Table 74 *Table for computing the ICCL parameter CommaInitializer [Neu98]*

10.2.6 Matrix Expression DSL

We will only consider matrix expressions containing matrix-matrix addition, subtraction, and multiplication, e.g. A+B+C-D*E or (A-B)*(C+D+E*F). Of course, the operand matrices have to have compatible dimensions (see Section 10.1.2.2.1.4.2).

We can assign a matrix expression to another matrix, e.g.:

R= A+B+C-D*E.

In the following sections, we discuss the question of what code should be generated for matrix expressions and matrix assignment.

10.2.6.1 Evaluating Matrix Expressions

Let us assume that A, B, C, D, and E are matrices and that they are compatible to be used in the following assignment statement:

E= (A + B) * (C + D)

There are two principal ways to compute this assignment [Neu98]:

1. with intermediate results (i.e. with temporaries)
   1.1. temp1= A + B
   1.2. temp2= C + D
   1.3. temp3= temp1 * temp2
   1.4. E= temp3
2. without temporaries (i.e. lazy)

   All elements of E are computed one after another from the corresponding elements of A, B, C, and D, i.e.

   e_{11}= $(a_{11} + b_{11}) * (c_{11} + d_{11}) +$
   $(a_{12} + b_{12}) * (c_{21} + d_{21}) +$
   ⋮
   ⋮
   e_{21}= …
   ⋮
   ⋮
   e_{mn}= …

Each of these two approaches have their advantages and disadvantages. The first approach is simple to implement using overloaded binary operators. Unfortunately, the initialization of the temporaries, the separate loops for each binary operation, and the final assignment incur a significant overhead.

The second (i.e. lazy) approach is particularly useful if we want to compute only some of the elements of a matrix expression (remember that the first approach computes all elements of each subexpression). Furthermore, it is also superior if the elements of the argument matrices (or subexpressions) are accessed only once during the whole computation. This is the case for expressions (or subexpressions) consisting of matrix additions only. In this case, approach two allows us to evaluate such expression very efficiently: we use two nested loops to iterate over the nonzero region of the resulting matrix and in each iteration we assign to the current element of the resulting matrix the sum of the corresponding elements from all the argument matrices. Thus, we do not need any temporaries and extra loops as in the first approach.[162]

Unfortunately, the second approach is inefficient for matrix multiplication since matrix multiplication accesses the elements of the argument matrices more than once. This is illustrated in Figure 174.

$$\begin{pmatrix} a_{11} & a_{21} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \bullet \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22} & a_{11}b_{13}+a_{12}b_{23} \\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22} & a_{21}b_{13}+a_{22}b_{23} \\ a_{31}b_{11}+a_{32}b_{21} & a_{31}b_{12}+a_{32}b_{22} & a_{31}b_{13}+a_{32}b_{23} \end{pmatrix}$$

Figure 174 *Elementwise computation of the multiplication of two sample matrices*

The elementwise computation of the multiplication of the two sample matrices requires each element of the argument matrices to be accessed three times. If the argument matrices are actually matrix expressions, each element of the argument expressions will be computed three times (asumming that both matices do not have nay special shape), which causes an unnecessary overhead.

The negative behavior of the lazy approach in the case of multiplication can be addressed in two ways:

- *Lazy with temporaries*: Whenever one of the arguments to a matrix multiplication is an expression, we create a temporary matrix and assign the expression to it. In effect, we use the lazy approach for matrix addition only.

- *Lazy with cache*: Instead of creating the full temporary for the expression arguments to a matrix multiplication at once (i.e. as above), we can create a cache matrix and compute any element of the argument expression on first access only. In other words, we use the lazy approach for both addition and multiplication and avoid recomputing elements of subexpressions by storing them in a cache.

Both approaches avoid the creation of temporaries for assignment (e.g. temp3 in our original example) and for arguments to matrix addition. However, they also have significant differences. Compared to the first one, the second approach has the overhead of the caching (i.e. store the computed element on first access and do the extra check whether the element has been already computed or not on each access). The effect of this overhead depends on many factors such as element type (i.e. precision, real or complex), number of operation per access, access time for the storage formats used. On the other hand, the lazy-with-cache approach is superior if we want to compute only some but not all of the expression elements. For example, if we are only interested in elements (1,1), (1,2), and (1,3) of the result matrix in Figure 174 and the argument matrices

are actually expressions, the lazy-with-cache approach is likely to be faster than the lazy-with-temporaries one.

In the following discussion, we will only consider the lazy-with-cache approach.

10.2.6.2 Implementing the Lazy Approach

An elegant thought model for the lazy approach is to think of the matrix expressions (and also its subexpressions) as objects. If you need to know some element of the expression, you ask the expression for it. An expression object implements only one matrix operation (i.e. addition, subtraction, or multiplication) itself and delegates the rest of the work to its subexpressions.

For example, consider the matrix expression A+B+C. Its object-oriented interpretation is shown in Figure 175. We have one addition expression object pointing to A and to another addition expression objects, which in turn points to B and C. When we ask the top-level addition expression for the element (i,j) by sending it the message getElement(i,j), it executes its getElement() method code shown in the box. This execution involves sending getElement(i,j) to its operands. The same happens for the second addition expression. Finally, each of the argument matrices gets the message getElement(i,j).

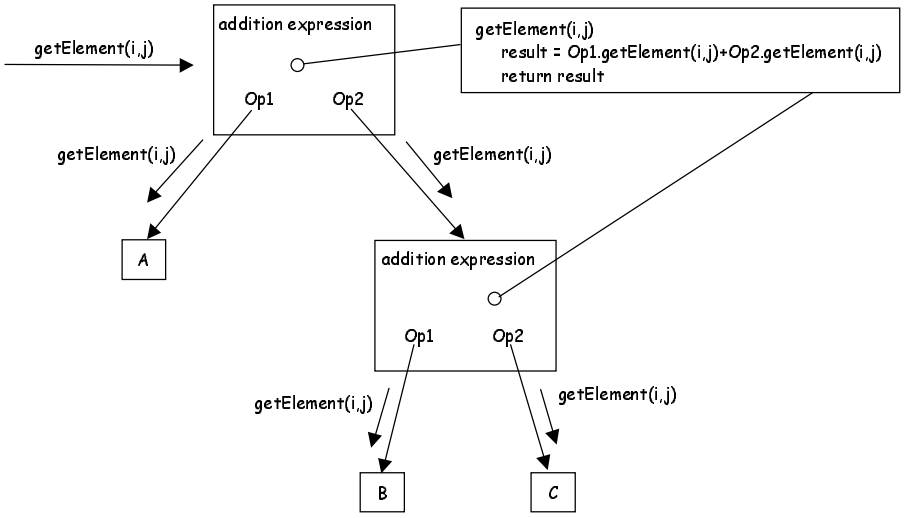

Figure 175 *Computing A+B+C using expression objects*

If the calls to getElement() are statically bound and we use inlining, the code generated for the top-level method getElement() looks like this:

```
A.getElement(i,j) + B.getElement(i,j) + C.getElement(i,j)
```

Now, assume that we want to assign our expression A+B+C to D:

```
D = A+B+C
```

This scenario is illustrated in Figure 176. We send the message "=" to D with A+B+C as its parameter. The assignment code (shown in the box) iterates through all elements of D and assigns each of them the value computed by sending getElement() to the expression.

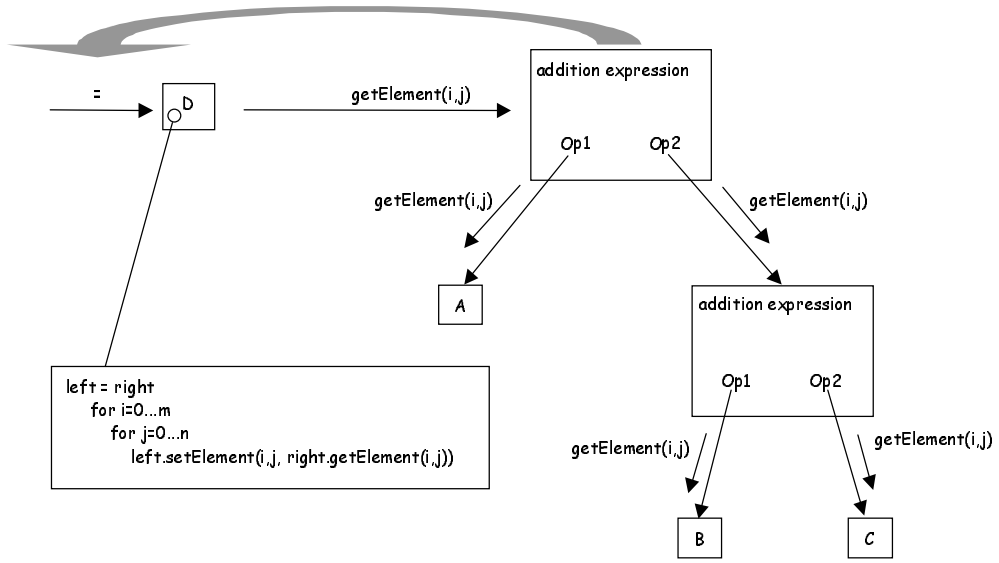

Figure 176 *Computing D = A+B+C using expression objects*

If we use inlining, the code generated for D = A+B+C will look like this:

```
for i=0...m
   for j=0...n
      D.setElement(A.getElement(i,j) + B.getElement(i,j) + C.getElement(i,j))
```

This is as efficient as we can get for general matrices.

The expression objects for the matrix expression (A+B)*(C+D) are shown in Figure 177. The getElement() method of the multiplication expression accesses Op1 and Op2 through caching wrappers. They make sure that no element of the addition expressions is computed more than once.

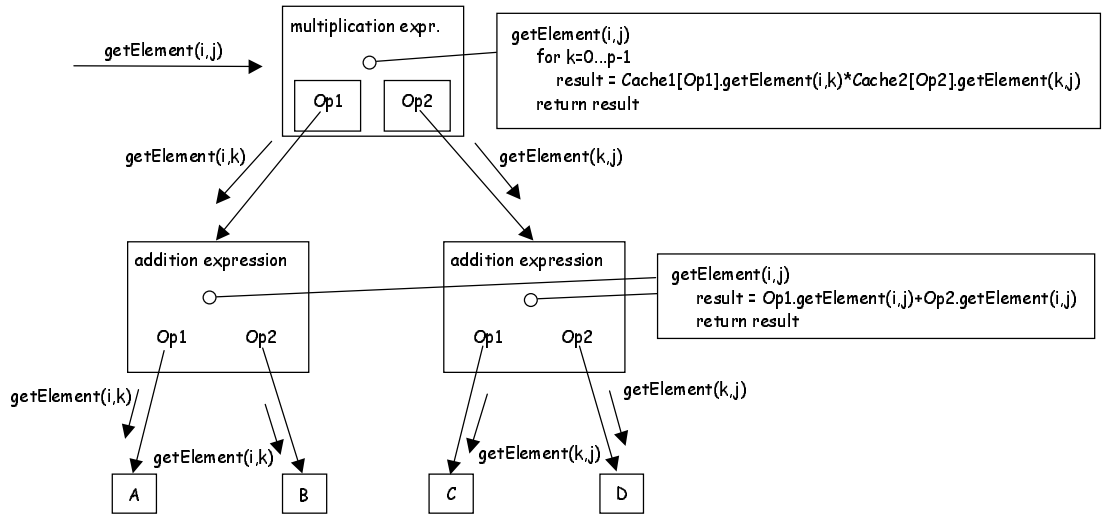

Figure 177 *Computing (A+B)*(C+D) using expression objects*

The sample code for the getElement() method and the assignment operation did not assume any special shape of the argument matrices. However, if the arguments have some special shape, we can usually select a faster implementation for getElement() and the assignment.

For example, if the left operand of a matrix multiplication expression has diagonal shape, we can use the following implementation of getElement():

```
getElement(i,j)
   result = Op1.getElement(i,i) * Op2.getElement(i,j)
   return result
```

Please note that this code is much simpler than the one in Figure 177. First, we do not need a loop. Second, we do not even need the caching wrappers since no element of the operands is accessed more than once.

Similarly, if the resulting shape of an expression is smaller than rectangular (e.g. triangular or diagonal), we only need to iterate over the nonzero region of the expression when assigning it to another matrix. In general, we can implement the assignment by initializing the target matrix and iterating over the nonzero part of the expression and assigning the computed elements to the corresponding elements of the matrix.

Thus, the job of the generator for the Matrix Expression DSL will be to select the appropriate code for the assignments and the calls to getElement() based on the computed type of the argument expressions. The operations themselves will be glued statically by inlining.

Next we will specify the available variant implementation of assignment and getElement() and when to select which. Finally, in Section 10.2.7, we specify how to compute the result type of a matrix expressions.

10.2.6.3 Assignment

The available assignment variants are listed in Table 75. A variant is selected based on the shape and the density of the source matrix in an assignment (see Table 76).

Assignment Component	Explanation
ZeroAssignment	Implements the assignment of a zero matrix by calling initElements on the target matrix.
DiagAssignment	Implements the assignment of a diagonal matrix (see Section 10.2.6.2).
RectAssignment	Implements the assignment of a rectangular matrix (see Section 10.2.6.2).
BandAssignment	Implements the assignment of a band matrix.
SparseAssignment	Implements the assignment of a sparse matrix. The algorithm uses iterators to iterate through the nonzero elements of the sparse source matrix.

Table 75 *Available assignment algorithms*

Shape	Density	Assignment
zero	*	ZeroAssignment
ident scalar diag	*	DiagAssignment
rect	dense	RectAssignment
*	sparse	SparseAssignment
*	*	BandAssignment

Table 76 *Selecting the assignment algorithm*

10.2.6.4 getElement()

The implementation of getElement() depends on the shape of the arguments. The selection table for matrix addition is shown in Table 77 and for matrix multiplication in Table 78.

Shape1	Shape2	GetElement
zero	zero	ZeroGetElement
ident	zero	IdentGetElement
zero	ident	IdentGetElement
zero	*	GetRightElement
*	zero	GetLeftElement
scalar ident	scalar ident	ScalarAddGetElement
rect	rect	RectGetAddElement
*	*	BandAddGetElement

Table 77 *Selecting the implementation of getElement() for matrix addition*

Shape1	Shape2	GetElement
zero	*	ZeroGetElement
*	zero	ZeroGetElement
ident	ident	IdentGetElement
ident	*	GetRightElement
*	ident	GetLeftElement
scalar diag	scalar diag	DiagMultiplyGetElement
scalar diag	*	DiagXMultiplyGetElement
*	scalar diag	XDiagMultiplyGetElement
rect	rect	RectMultiplyGetElement
*	*	BandMultiplyGetElement

Table 78 *Selecting the implementation of getElement() for matrix addition*

10.2.7 Computing Result Types of Operations

In order to be able to declare variables for the intermediate results (or for the caches) and also select the optimal implementations of getElement() and assignment during the code generation for matrix expressions, we have to be able to compute the result type of the operations based on the operation argument types. Since a flat configuration represents an abstract and complete description of a matrix type (see Section 10.2.3.5), we do the result type computation at the level of flat configurations. All we have to do is to retrieve the flat configurations from the configuration repositories of the argument matrix types and give them to a metafunction which will compute the resulting flat configuration. Next, the resulting flat configuration can be given to the matrix generator in order to generate the resulting matrix type.

The goal of this section is to specify how to compute the resulting flat configuration from the flat configurations of the arguments for different operations. We only need to specify the mapping for addition and multiplication since addition and subtraction use the almost same mapping.[163]

The computation of the mathematical properties of the resulting matrix can be based on the mathematical theory of matrices. For example, the shape of a matrix resulting from the multiplication or the addition of two lower-triangular matrices is also lower triangular. On the other hand, the result of combining properties such as error checking, optimization flag, etc., is not that obvious. Our current strategy for the latter kind of properties is to return the same property if the argument properties are equal or return unspecified otherwise (see Table 83). If we give a flat configuration with some unspecified features to the matrix generator, it will assume the feature defaults described in Sections 10.2.3.3 and 10.2.3.4. For example, if OptFlag of both argument matrices is speed, OptFlag of the resulting matrix will also be speed. If, on the other hand, one of them is speed and the other one space, then we assume the resulting OptFlag to be unspecified_DSL_feature. According to Table 42, the generator will use space for the resulting OptFlag.

It is worth noting that the result type computation specifies static, domain-specific type inference. This is so since the computation operates on static descriptions of properties of the matrices. Indeed, this inference and the flat configurations define a domain-specific type system for matrices.

Now, we specify the result computation for each of the flat configuration DSL parameters for addition and multiplication. Please note that the following dependency tables give one possible solution and other solutions could be equally good or better. We certainly do not use all possible mathematical dependencies between the features and the choices for the more arbitrary features

might be disputable. Nevertheless, these functions appear to be usable and, most importantly, they can be extended and improved any time later.

10.2.7.1 Element Type and Index Type

ElementType and IndexType are numeric types. Thus, the resulting type has to be the larger one. In C++, we achieve this with a type promotion metafunction which returns the type with a larger exponent or, in case the exponents are equal, the one with larger precision. The code is shown in Figure 178. The C++ code uses the standard C++ traits template numeric_limits<>, which describes the properties of built-in types.

```
#include <limits>
using namespace std;

template<class A, class B>
struct PROMOTE_NUMERIC_TYPE
{
   typedef IF<
      numeric_limits<A>::max_exponent10 < numeric_limits<B>::max_exponent10
      ||
      (numeric_limits<A>::max_exponent10==numeric_limits<B>::max_exponent10
         &&
       numeric_limits<A>::digits < numeric_limits<B>::digits),

      B,
      A>::RET RET;
};
```

Figure 178 *C++ meatafunction for promoting nummeric types*

10.2.7.2 Shape

Shape1	Shape2	Shape Result
zero	*	zero
*	zero	zero
ident scalar	(shape2)	=(shape2)
(shape1)	ident scalar	=(shape1)
diag	symm	rect
symm	diag	rect
diag	(shape2)	=(shape2)
(shape1)	diag	=(shape1)
bandDiag	bandDiag	bandDiag
lowerBandTriang	lowerBandTriang	lowerBandTriang
upperBandTriang	upperBandTriang	upperBandTriang
lowerTriang lowerBandTriang	lowerTriang lowerBandTriang	lowerTriang
upperTriang upperBandTriang	upperTriang upperBandTriang	upperTriang
*	*	rect

Table 79 *Resulting Shape for multiplication [Neu98]*

Shape1	Shape2	Shape Result
ident	ident	scalar
zero ident scalar diag	(shape2)	=(shape2)
(shape1)	zero ident scalar diag	=(shape1)
symm	symm	symm
bandDiag	bandDiag	bandDiag
lowerBandTriang	lowerBandTriang	lowerBandTriang
upperBandTriang	upperBandTriang	upperBandTriang
lowerTriang lowerBandTriang	lowerTriang lowerBandTriang	lowerTriang
upperTriang upperBandTriang	upperTriang upperBandTriang	upperTriang
*	*	rect

Table 80 *Resulting Shape for addition [Neu98]*

10.2.7.3 Format

Shape Result	Format1	Format2	Format Result
rect	vect	vect	unspecified_DSL_feature
rect	DIA	DIA	unspecified_DSL_feature
rect	SKY	SKY	unspecified_DSL_feature
*	(value)	(value)	=(value)
*	*	*	unspecified_DSL_feature

Table 81 *Resulting Format (addition and multiplication)*

10.2.7.4 Density

Density1	Density2	Density Result
sparse	sparse	sparse
*	*	dense

Table 82 *Resulting Density (addition and multiplication) [Neu98]*

10.2.7.5 Malloc, OptFlag, BoundsChecking, CompatChecking, MallocErrChecking, DictFormat, and ArrOrder

Feature1	Feature2	Result
(value)	(value)	=(value)
*	*	unspecified_DSL_feature

Table 83 *General formula for computing results of non-mathematical properties (addition and multiplication) [Neu98]*

10.2.7.6 Size and HashWidth

Size1	Size2	Size Result
(value1)	(value2)	=Max((value1),(value2))

Table 84 *Resulting Size (addition and multiplication) [Neu98]*

HashWidth1	HashWidth2	HashWidth Result
(value1)	(value2)	=Max((value1),(value2))

Table 85 *Resulting HashWidth (addition and multiplication) [Neu98]*

10.2.7.7 Rows and RowsNumber (Multiplication)

Rows1	Order1	Rows Result	RowsNumber Result
stat_val	*	stat_val	=RowsNumber1
*	stat_val	stat_val	=OrderNumber1
*	*	dyn_val	unspecified_DSL_feature

Table 86 *Resulting RowsNumber (multiplication) [Neu98]*

10.2.7.8 Cols and ColsNumber (Multiplication)

Cols2	Order2	Cols Result	ColsNumber Result
stat_val	*	stat_val	=ColsNumber2
*	stat_val	stat_val	=OrderNumber2
*	*	dyn_val	unspecified_DSL_feature

Table 87 *Resulting ColsNumber (multiplication) [Neu98]*

10.2.7.9 Order and OrderNumber (Multiplication)

Order1	Order2	Order Result	OrderNumber Result
stat_val	stat_val	stat_val	=OrderNumber1
*	*	dyn_val	unspecified_DSL_feature

Table 88 *Resulting OrderNumber (multiplication) [Neu98]*

10.2.7.10 Diags and DiagsNumber (Multiplication)

Diags1	Diags2	Diags Result
stat_val	stat_val	stat_val
*	*	dyn_val

Table 89 *Resulting Diags (multiplication) [Neu98]*

Shape	Diags Result	DiagsNumber Result
lowerBandTriang upperBandTriang bandDiag	stat_val	=DiagsNumber1 + DiagsNumber2 - 1
*	*	unspecified_DSL_feature

Table 90 *Resulting DiagsNumber (multiplication) [Neu98]*

10.2.7.11 ScalarValue (Multiplication)

ScalarValue1	ScalarValue2	ScalarValue Result	ScalarValueNumber Result
stat_val	stat_val	stat_val	=ScalarValueNumber1 * ScalarValueNumber2
*	*	dyn_val	unspecified_DSL_feature

Table 91 *Resulting ScalarValue (multiplication) [Neu98]*

10.2.7.12 Ratio and Growing (Multiplication)

We do not compute the result of Ratio and Growing but rather set them to unspecified_DSL_feature. This will cause the matrix generator to assume the default values for both parameters.

10.2.7.13 Rows, RowsNumber, Cols, and ColsNumber (Addition)

Rows1	Rows2	Rows Result	RowsNumber Result
stat_val	*	stat_val	=RowsNumber1
*	stat_val	stat_val	=RowsNumber2
*	*	dyn_val	unspecified_DSL_feature

Table 92 *Resulting RowsNumber (addition) [Neu98]*

Cols1	Cols2	Cols Result	ColsNumber Result
stat_val	*	stat_val	=ColsNumber1
*	stat_val	stat_val	=ColsNumber2
*	*	dyn_val	unspecified_DSL_feature

Table 93 *Resulting ColsNumber (addition) [Neu98]*

10.2.7.14 Order and OrderNumber (Addition)

Order1	Order2	Order Result	OrderNumber Result
stat_val	*	stat_val	=OrderNumber1
*	stat_val	stat_val	=OrderNumber2
*	*	dyn_val	unspecified_DSL_feature

Table 94 *Resulting OrderNumber (addition) [Neu98]*

Additionally, we have to treat the following case as an exception:

Shape == rect and Order == stat_val

In this case, the values for Rows, Cols, RowsNumber, and ColsNumber computed based on other tables have to be overridden as follows:

Rows= stat_val
Cols= stat_val
RowsNumber= OrderNumber
ColsNumber= OrderNumber

10.2.7.15 Diags and DiagsNumber (Addition)

Diags1	Diags2	Diags Result
stat_val	stat_val	stat_val
*	*	dyn_val

Table 95 *Resulting Diags (addition) [Neu98]*

Shape	Diags	DiagsNumber Result
lowerBandTriang upperBandTriang bandDiag	stat_val	=Max(DiagsNumber1, DiagsNumber2)
*	*	unspecified_DSL_feature

Table 96 *Resulting DiagsNumber (addition) [Neu98]*

10.2.7.16 ScalarValue (Addition)

ScalarValue1	ScalarValue2	ScalarValue Result	ScalarValueNumber Result
stat_val	stat_val	stat_val	=ScalarValueNumber1 + ScalarValueNumber2
*	*	dyn_val	unspecified_DSL_feature

Table 97 *Resulting ScalarValue (addition) [Neu98]*

10.2.7.17 Ratio and Growing (Addition)

Ratio1	Ratio2	Ratio Result
(value1)	(value2)	=ArithmeticAverage((value1),(value2))

Table 98 *Resulting Ratio (addition)*

Growing1	Growing2	Growing Result
(value1)	(value2)	=ArithmeticAverage((value1),(value2))

Table 99 *Resulting Growing (addition)*

10.3 Domain Implementation

This section covers the implementation of the matrix component specified in the domain implementation section (i.e. Section 10.2). First, we discuss how to implement the component in C++. For demonstration purpose, we will show the implementation of a very small matrix component, which nonetheless, allows us to explain most of the relevant implementation techniques. The full C++ implementation of the matrix component is described in Section 10.3.1.8.

In Section 10.3.2, we summarize the experience we made during the implementation of a subset of the matrix component in the Intentional Programming System.

10.3.1 C++ Implementation

10.3.1.1 Architecture Overview

The architecture of the C++ implementation of the matrix component is shown in Figure 179. The pipeline on the left compiles matrix configuration expressions, e.g.

```
matrix<double, structure<rect<dyn_val<>, dyn_val<>, CSR<> >, sparse<> > >
```

whereas the pipeline on the right compiles matrix expressions, e.g.

```
M5 = M1+M2*(M3-M4);
```

The compilation of a matrix configuration expression involves parsing, computing defaults for the unspecified features, assembling the GenVoca-like components according to the feature values, and storing the feature values in the configuration repository, which becomes part of the produced matrix type. Features are encoded as types (numbers are encoded using enumeration types). We group them by putting them into a class (or struct) as its member types. This is exactly how the configuration repository is implemented.

A matrix expression is parsed using *expression templates* [Vel95]. We will explain this idea in Section 10.3.1.7.1. The parsing result is a tree of expression objects (or rather the type of the tree to be instantiated at runtime) much in the style shown in Figure 175 and Figure 177. We also need to compute the matrix type of the expression and of all its subexpressions, so that the code generation templates can select the appropriate implementations for the assignment and the `getElement()` methods (see Section 10.2.6.2) and ask the component assembler to generate the matrix types for the intermediate results (or matrix caches; see Section 10.2.6.1) if any needed.[164]

Any part of the architecture requiring compile-time execution (e.g. computing feature defaults, assembling components, computing result types) is implemented as template metafunctions (we discussed template metafunctions in Chapter 8).

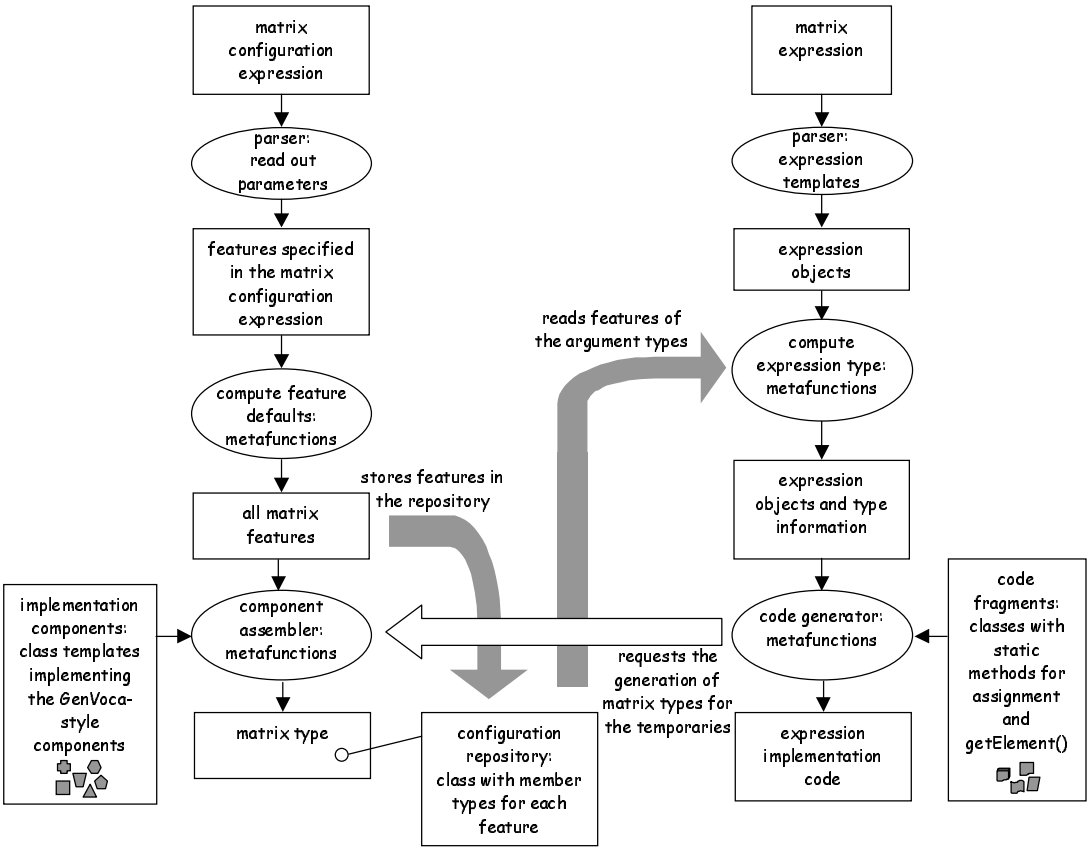

Figure 179 *Architecture of the C++ implementation of the matrix component*

The whole matrix component is implemented as a C++ template library. The compilation of a client program including this library is done entirely by a C++ compiler (i.e. we neither use preprocessors nor any generation tools). This is possible since template metaprograms are interpreted by the compiler at compile time. Figure 180 illustrates the compilation of some demo client code using the matrix library.

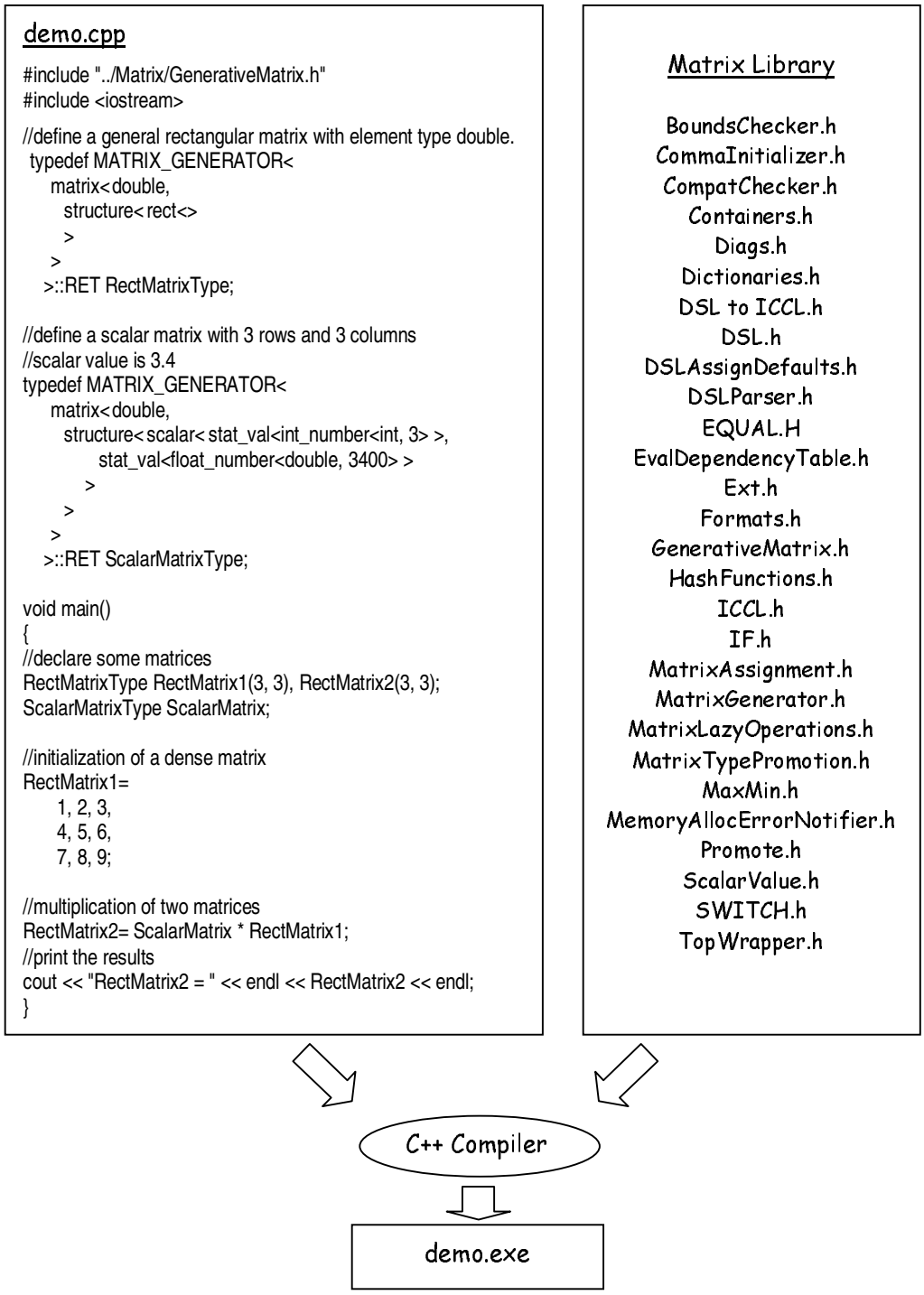

Figure 180 *Compilation scenario of a matrix demo program*

The following sections describe the C++ implementation techniques used to implement the generator. We demonstrate theses techniques using a smaller matrix component, which we describe as next.

10.3.1.2 Overview of the Demo Implementation

The C++ implementation of the demo matrix component consists of a number of C++ modules listed in Table 100. The modules are grouped into five categories. The following sections cover the implementation of each module in the order they are listed in the table. Each section contains the full module source code. The beginning of a new module is marked by a gray side box indicating its name. The source code itself is marked by a vertical ruler to its left. This helps to distinguish it from explanations and code examples (the latter do not have the vertical ruler).

Category	Contents	Modules	Contents	Section
Matrix Configuration DSL	specification and implementation of the Matrix Configuration DSL	DSL.h		10.3.1.3
Matrix ICCL	specification and implementation of the matrix implementation components	ICCL.h	specification of the matrix ICCL	10.3.1.4
		Containers.h	basic containers	10.3.1.4.1
		Formats.h	formats	10.3.1.4.2
		BoundsChecker.h	bounds checker	10.3.1.4.3
		TopWrapper.h	matrix top wrapper	10.3.1.4.4
		CommaInitializer.h	comma initializer	
Matrix Configuration Generator	implementation of the matrix configuration generator	MatrixGenerator.h	matrix configuration generator	10.3.1.5
		DSLParser.h	parse matrix configuration DSL	10.3.1.5.1
		DSLAssignDefaults.h	assign defaults to unspecified DSL features	10.3.1.5.2
		AssembleComponents.h	assemble implementation components	10.3.1.5.3
Matrix Operations	implementation of the matrix operations (Matrix Expression DSL)	MatrixOperTemplates.h	operator templates for the matrix operations	10.3.1.7.1
		MatrixExprTemplates.h	matrix operation expression class templates	10.3.1.7.2
		MatrixCache.h	matrix cache used for matrix multiplication	10.3.1.7.3
		GetElement.h	different implementations of getElement() method for addition and multiplication expressions of different shapes	10.3.1.7.4
		ComputeResultType.h	compute the matrix result type of matrix operations	10.3.1.7.5
		Assignment.h	different implementations of assignment for different matrix shapes	10.3.1.7.6
Auxiliary		GenerativeMatrix.h	general include file	
		Promote.h	promoting numeric types	10.2.7.1
		IF.h	meta if	8.2
		SWITCH.h	meta switch	8.13
		equal.h	auxiliary metafunction to be used with meta if	

Table 100 *Overview of the C++ implementation of the demo matrix component*

10.3.1.3 Matrix Configuration DSL

The matrix configuration DSL that we are going to implement is shown in Figure 181 as a feature diagram. This is an extremely simplified version of the full matrix configuration DSL described in Section 10.2. The DSL allows us to specify matrix element type, matrix shape, storage format, whether to optimize for speed or space, whether to do bounds checking or not, and index type.

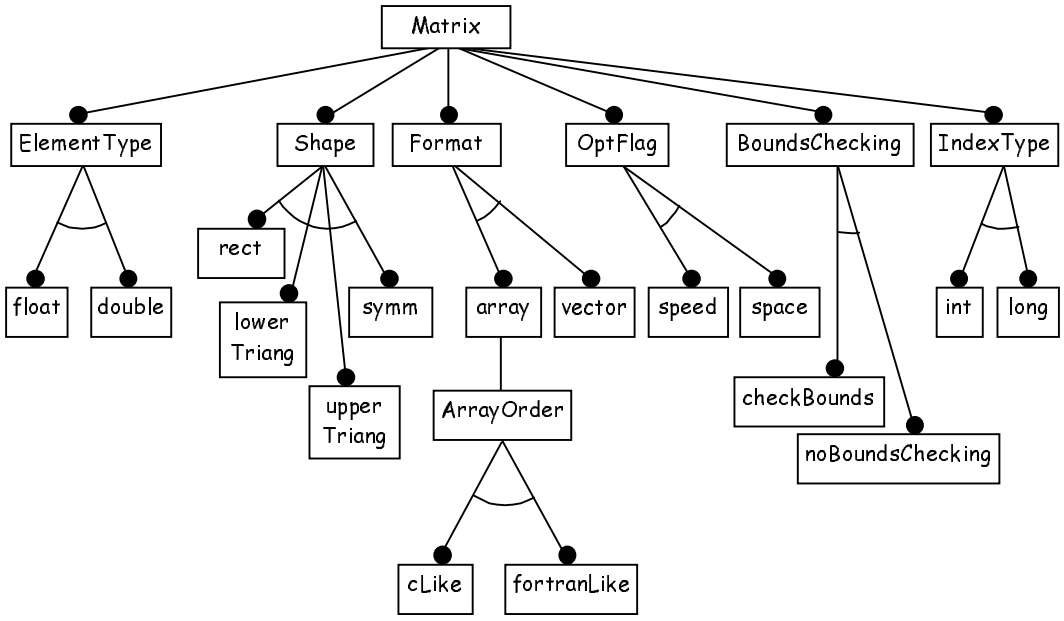

Figure 181 *Configuration DSL of the demo matrix component (Please note that not all possible element and index types are shown; cf. Figure 182)*

The grammar specification of the configuration DSL is given in Figure 182.

Matrix:	matrix[ElementType, Shape, Format, OptFlag, BoundsChecking, IndexType]
ElementType:	float | double | long double | int | long | ...
Shape:	rect | lowerTriang | upperTriang | symm
Format:	array[ArrayOrder] | vector
ArrayOrder:	cLike | fortranLike
OptFlag:	speed | space
BoundsChecking:	checkBounds | noBoundsChecking
IndexType:	char | short | int | long | unsigned int | ...

Figure 182 *Grammar specification of the demo matrix configuration DSL*

We implement the DSL using nested class templates in the module DSL.h.

First thing to do is to copy the grammar definition in Figure 182 and paste it in DSL.h as a comment. Next, under each grammar production, we directly type in the corresponding template declarations:

Module: DSL.h

```
namespace MatrixDSL {

//Matrix: matrix[ElementType, Shape, Format, OptFlag, BoundsChecking, IndexType ]
template<
  class ElementType, class Shape, class Format, class OptFlag,  class BoundsChecking, class IndexType
>struct matrix;

//ElementType :  float | double | long double | short | int | long | unsigned short |
//         unsigned int | unsigned long
//built-in types – nothing to declare

//Shape:   rect | upperTriang | lowerTriang |  symm
template<class dummy>struct rect;
template<class dummy>struct lower_triang;
template<class dummy>struct upper_riang;
template<class dummy>struct symm;
```

Alternatively, we could have implemented the various shape values simply as structs rather than struct templates. However, by implementing them as templates we keep the DSL more extendible. This is so since we could add subfeatures to the values without having to change existing client code. All we have to do is to make sure that each template parameter has a default. In this case, if a client writes

```
rect<>
```

we really do not know how many parameters the rect template has.

We continue with the remaining grammar productions in a similar fashion:

```
//Format:  array[ArrayOrder] | vector
template<class ArrOrder>struct array;
template<class dummy>struct vector;


//ArrOrder:    cLike | fortranLike
template<class dummy>struct c_like;
template<class dummy>struct fortran_like;


//OptFlag :    speed | space
template<class dummy>struct speed;
template<class dummy>struct space;


//BoundsChecking : checkBounds | noBoundsChecking
template<class dummy>struct check_bounds;
template<class dummy>struct no_bounds_checking;


//IndexType : char | short | int | long | unsigned char | unsigned short |
//        unsigned int | unsigned long | signed char


//type denoting "value unspecified"
struct unspecified_DSL_feature;
```

The last struct is used as a feature value denoting “value unspecified”. We can use this feature in matrix configuration expressions if we do not want to specify some feature in the middle of a parameter list, e.g.

```
matrix<float,lower_triang,unspecified_DSL_feature,speed>
```

As you have probably already anticipated, we do not have to specify the last two parameters (i.e. bounds checking and index type) since the matrix template defines appropriate defaults.

Now, let us take a look at the implementation of the DSL features. We start with the implementation of unspecified_DSL_feature that we have just mentioned above. As stated, it is used to denote “value unspecified”. However, this is not its only purpose. We also use it as the superclass for all other DSL feature values. This has the following reason: unspecified_DSL_feature defines identification numbers for all DSL features. Identification numbers, or IDs, are used to test types for equality. This is necessary since there is no other means in C++ to do it.

First, we define unspecified_DSL_feature:

```
struct unspecified_DSL_feature
{
  enum {
    unspecified_DSL_feature_id = -1,

    // IDs of Shape values
    rect_id,
    lower_triang_id,
    upper_triang_id,
```

```
    symm_id,

    //IDs of Format values
    array_id,
    vector_id,

    //IDs of ArrOrder values
    c_like_id,
    fortran_like_id,

    //IDs of OptFlag values
    speed_id,
    space_id,

    //IDs of BoundsChecking values
    check_bounds_id,
    no_bounds_checking_id,

    //my own ID
    id=unspecified_DSL_feature_id };
};
```

Here is the implementation of the first DSL production:

```
//Matrix: matrix[ElementType, Shape, Format, OptFlag, BoundsChecking, IndexType ]
template<
  class ElementType = unspecified_DSL_feature,
  class Shape = unspecified_DSL_feature,
  class Format = unspecified_DSL_feature,
  class OptFlag = unspecified_DSL_feature,
  class BoundsChecking = unspecified_DSL_feature,
  class IndexType = unspecified_DSL_feature >
struct matrix
{
  typedef ElementType elementType;
  typedef Shape shape;
  typedef Format format;
  typedef OptFlag optFlag;
  typedef BoundsChecking boundsChecking;
  typedef IndexType indexType;
};
```

Please note that we use unspecified_DSL_feature as the default value for each parameter. Of course, we will assign some more useful default values later in the generator. The reason for not assigning the defaults here is that we want to assign all defaults (i.e. both direct and computed) in one place, which will be the matrix generator.

The next thing to point out is that matrix<> defines each parameter type as its member type. We say that it *publishes* its parameters. This is so since now we can access its parameters as follows:

```
matrix<Foo1,Foo2>::shape //this is equivalent to Foo2
```

The final detail is the reason why we use a struct and not a class: all members of matrix are public and by using a struct we do not have to write the extra access modifier public:.

The next production consists of a number of alternative values:

```
//Shape :    rect | lowerTriang | upperTriang | symm
template<class dummy = unspecified_DSL_feature>
struct rect : unspecified_DSL_feature
{ enum { id=rect_id };
};

template<class dummy = unspecified_DSL_feature>
struct lower_triang : unspecified_DSL_feature
{ enum { id=lower_triang_id };
};
```

```
template<class dummy = unspecified_DSL_feature>
struct upper_triang : unspecified_DSL_feature
{ enum { id=upper_triang_id };
};

template<class dummy = unspecified_DSL_feature>
struct symm : unspecified_DSL_feature
{ enum { id=symm_id };
};
```

Each "dummy" parameter has unspecified_DSL_feature as its default. As you remember, the purpose of this parameter was to make values without subfeatures templates, so that new subfeatures can be added without having to modify existing client code. Each value "publishes" its ID using an enum declaration. The initialization values for the IDs were defined in unspecified_DSL_feature, the superclass of all feature values. The following example demonstrates the use of IDs:

```
typedef rect<> Shape1;
typedef upper_triang<> Shape2;
typedef upper_triang<> Shape3;

cout << (Shape1::id == Shape2::id);  //prints: 0
cout << (Shape2::id == Shape3::id);  //prints: 1
```

The approach with the IDs allows us for even a finer testing than just type equality: we can test if two types were instantiated from the same class template even if the types are not equal (i.e. different parameters were used):

```
typedef array<c_like<> > Format1; //array<> and c_like are defined below
typedef array<fortran_like<> > Format2; //fortran_like<> is defined below

cout << (Format1::id == Format2::id);   //prints: 1
```

The remaining DSL features are specified in a similar way:

```
//Format   :  array[ArrOrder] | vector
template<class ArrOrder= unspecified_DSL_feature>
struct array : unspecified_DSL_feature
{ enum {id= array_id};
  typedef ArrOrder arr_order;
};

template<class dummy= unspecified_DSL_feature>
struct vector : unspecified_DSL_feature
{ enum {id= vector_id};
};


//ArrOrder:    cLike | fortranLike
template<class dummy = unspecified_DSL_feature>
struct c_like : unspecified_DSL_feature
{ enum { id= c_like_id };
};

template<class dummy= unspecified_DSL_feature>
struct fortran_like : unspecified_DSL_feature
{ enum {id= fortran_like_id};
};


//OptFlag :    speed | space
template<class dummy = unspecified_DSL_feature>
struct speed : unspecified_DSL_feature
{ enum { id=speed_id };
};

template<class dummy = unspecified_DSL_feature>
struct space : unspecified_DSL_feature
{ enum { id=space_id };
};
```

```
//BoundsChecking : checkBounds | noBoundsChecking
template<class dummy = unspecified_DSL_feature>
struct check_bounds : unspecified_DSL_feature
{ enum { id=check_bounds_id };
};

template<class dummy = unspecified_DSL_feature>
struct no_bounds_checking : unspecified_DSL_feature
{ enum { id=no_bounds_checking_id };
};

} //namespace MatrixDSL
```

This concludes the implementation of the matrix configuration DSL.

10.3.1.4 Matrix Implementation Components and the ICCL[165]

Figure 183 shows the GenVoca-like component architecture implementing the functionality scope specified in the previous section. The box at the bottom is the configuration repository (Config), which contains all the DSL features of a configuration and some extra types to be accessed by other components. The basic containers layer provides the containers for storing matrix elements: Dyn1DContainer (one-dimensional), Dyn2DCContainer (two-dimensional, row-wise storage) and Dyn2DFContainer (two-dimensional, column-wise storage). All containers allocate memory dynamically. On the top of the containers, we have three alternative format components: ArrFormat (used to store rectangular and triangular matrices), LoTriangVecFormat (used for lower triangular matrices), and UpTriangVecFormat (used for upper triangular matrices).[166] Symm is an optional wrapper for implementing the symmetry property of a matrix. BoundsChecker is another optional wrapper. It implements bounds checking. Finally, Matrix is the top-level wrapper of all matrices.

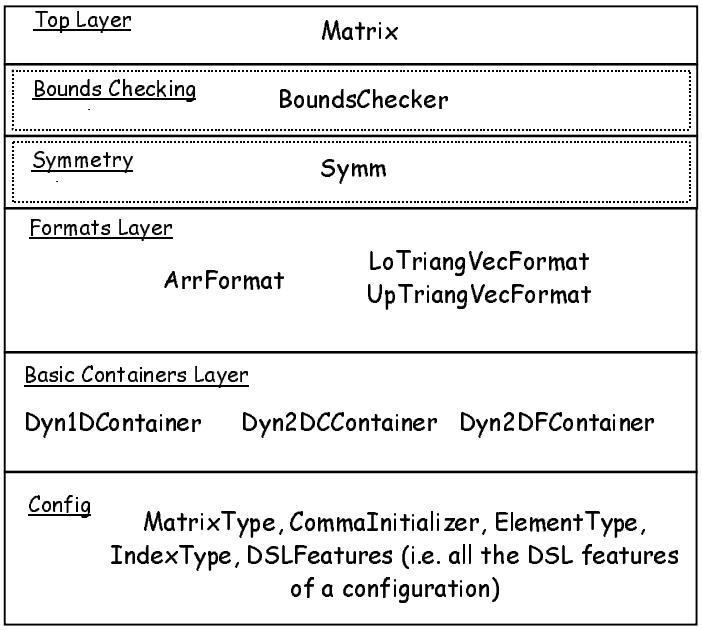

Figure 183 *Layers of the matrix implementation components*

The configurability of the matrix implementation components is specified by the ICCL grammar in Figure 184.

```
MatrixType:              Matrix[OptBoundsCheckedMatrix]
OptBoundsCheckedMatrix:  OptSymmetricMatrix | BoundsChecker[OptSymmetricMatrix]
OptSymmetricMatrix:      Format | Symm[Format]
Format:                  ArrFormat[Array] | LoTriangVecFormat[Vector] |
                         UpTriangVecFormat[Vector]
Array:                   Dyn2DCContainer[Config] | Dyn2DFContainer[Config]
Vector:                  Dyn1DContainer[Config]
Config:                  Config contains: MatrixType, CommaInitializer, ElementType, IndexType, and
                         DSLFeatures (i.e. all the DSL parameters of a configuration)
```

Figure 184 *ICCL grammar for the demo matrix package*

Each of the matrix implementation components will be implemented by a class template. But before we show the implementation of the components, we first need a file declaring all of the components. You can think of this file as the ICCL specification.

We start as in the case of the DSL specification by copying the ICCL grammar from Figure 184 into ICCL.h as a comment. We type in the component declarations under the corresponding ICCL production:

Module: ICCL.h

```
namespace MatrixICCL {

//Matrix :     Matrix [OptBoundsCheckedMatrix]
template<class OptBoundsCheckedMatrix>class Matrix;

//OptBoundsCheckedMatrix:   OptSymmetricMatrix | BoundsChecker[OptSymmetricMatrix]
template<class OptSymmetricMatrix>class BoundsChecker;

//OptSymmetricMatrix:     MatrixContainer | Symm[MatrixContainer]
template<class MatrixContainer>class Symm;

//Format:     ArrFormat[Array] | LoTriangVecFormat[Vector] | UpTriangVecFormat[Vector]
template< class Array>class ArrFormat;
template< class Vector>class LoTriangVecFormat;
template< class Vector>class UpTriangVecFormat;

//Array:    Dyn2DCContainer[Config] | Dyn2DFContainer[Config]
template<class Generator>class Dyn2DCContainer;
template<class Generator>class Dyn2DFContainer;
```

At this point, we need to explain one implementation detail. As you know, Config is the configuration repository which is always passed to the components in the bottom layer of a GenVoca architecture. But the two last template declarations take Generator as their parameter instead. This is not a problem, however, since Config is a member type of Generator. This detail has to do with some C++ compiler problems (specifically VC++5.0) which are not relevant here. Here are the remaining declarations:

```
//Vector:     Dyn1DContainer[Config]
template<class Generator> class Dyn1DContainer;

//CommaInitializer:     DenseCCommaInitializer | DenseFCommaInitializer
template<class MatrixType>class DenseCCommaInitializer;
template<class MatrixType>class DenseFCommaInitializer;

} //namespace MatrixICCL
```

We use CommaInitializer to provide matrix initialization by comma-separated lists of numbers.

Now we give you the implementation of the components. Each component group is treated in a separate section.

In case that you wonder where the Config is: Config is defined as a member class of the matrix component assembler discussed in Section 10.3.1.5.3.

10.3.1.4.1 Basic Containers

As shown in Figure 183, we need to implement three containers: Dyn1DContainer, Dyn2DCContainer, and Dyn2DFContainer. We start with Dyn1DContainer (in Containers.h):

Module: Containers.h

```
namespace MatrixICCL{

template<class Generator>
class Dyn1DContainer
{   public:
      typedef Generator::Config Config;
      typedef Config::ElementType ElementType;
      typedef Config::IndexType IndexType;
```

As already stated, Generator is expected to provide Config as its member type. Config, in turn, provides element type and index type. All matrix components can access these types in this fashion. Dyn1DContainer allocates the memory for storing its elements from the heap. The size is specified in the constructor:

```
    protected:
      IndexType size_;
      ElementType * pContainer;

    public:
      Dyn1DContainer(const IndexType& l)
        : size_(l)
      { assert(size()>0);
        pContainer = new ElementType [size()];
        assert( pContainer != NULL );
      }

      ~Dyn1DContainer() {delete [] pContainer;}

      void setElement(const IndexType& i, const ElementType& v)
      { checkBounds( i );
        pContainer[ i ] = v;
      }

      const ElementType & getElement(const IndexType& i) const
      { checkBounds( i );
        return pContainer[ i ];
      }

      const IndexType& size() const {return size_;}

      void initElements(const ElementType& v)
      { for( IndexType i = size(); i--; )
        setElement( i, v );
      }

    protected:
      void checkBounds(const IndexType& i) const
      { assert(i>=0 && i<size());
      }
};
```

Dyn2DCContainer is a two dimensional container storing its elements row-wise:

```
template<class Generator>
class Dyn2DCContainer
{
  public:
    typedef Generator::Config Config;
    typedef Config::ElementType ElementType;
    typedef Config::IndexType IndexType;

  protected:
```

```
    IndexType r_, c_;
    ElementType* elements_;
    ElementType** rows_;

  public:
    Dyn2DCContainer(const IndexType& r, const IndexType& c)
      : r_(r), c_(c)
    {
      assert(r_>0);
      assert(c_>0);

      elements_ = new ElementType[r*c];
      rows_ = new ElementType*[r];
      assert(elements_ != NULL);
      assert(rows_ != NULL);

      ElementType* p= elements_;
      for (IndexType i= 0; i<r; i++, p+= c) rows_[i]= p;
    }

    ~Dyn2DCContainer()
    { delete [] elements_;
      delete [] rows_;
    }

    void setElement(const IndexType& i, const IndexType& j, const ElementType& v)
    { checkBounds(i, j);
      rows_[i][j] = v;
    }

    const ElementType& getElement(const IndexType& i, const IndexType& j) const
    { checkBounds(i, j);
      return rows_[i][j];
    }

    const IndexType& rows() const { return r_; }
    const IndexType& cols() const { return c_; }

    void initElements(const ElementType& v)
    {    for(IndexType i = rows(); i--;)
         for(IndexType j = cols(); j--;)
           setElement(i, j, v);
    }

  protected:
    void checkBounds(const IndexType& i, const IndexType& j) const
    { assert(i>=0 && i<rows());
      assert(j>=0 && j<cols());
    }
};
```

Dyn2DFContainer is a two dimensional container storing its elements column-wise. We can easily derive its implementation from Dyn2DCContainer by inheritance. All we have to do is to override setElement() and getElement() to swap the argument indices and also override rows() and cols() to call the base cols() and rows(), respectively:

```
template<class Generator>
class Dyn2DFContainer : public Dyn2DCContainer<Generator>
{
  private:
    typedef Dyn2DCContainer<Generator> BaseClass;

  public:
    Dyn2DFContainer(const IndexType& r, const IndexType& c)
      : BaseClass(c, r)
    {}

    void setElement(const IndexType& i, const IndexType& j, const ElementType& v)
    {BaseClass::setElement(j, i, v);}
```

```
    const ElementType & getElement(const IndexType& i, const IndexType& j) const
    {return BaseClass::getElement(j, i);}

    const IndexType& rows() const {return BaseClass::cols();}
    const IndexType& cols() const {return BaseClass::rows();}
};

} //namespace MatrixICCL
```

10.3.1.4.2 Formats

For our demo matrix, we implement three formats: ArrFormat, LoTriangVecFormat, and UpTriangVecFormat. ArrFormat stores matrix elements directly in a two-dimensional container. This format is appropriate for storing rectangular and triangular matrices. In the latter case, only half of the allocated container memory is used. On the other hand, accessing the elements of a triangular matrix stored in ArrFormat does not involve any explicit index arithmetic. In case you prefer a more space-saving variant, you can use LoTriangVecFormat for lower triangular matrices and UpTriangVecFormat for upper triangular matrices. Each of the latter formats uses a one-dimensional container to store its elements. The elements of a symmetric matrix are stored the same way as the elements of a lower triangular matrix. The only difference is that, for a symmetric matrix, we wrap the format in Symm, which maps any access to the elements above the main diagonal to the elements of the lower half of the underlying format.

We start with ArrFormat. Since ArrFormat stores matrix elements in a two-dimensional container directly, there is hardly any difference between storing the elements of rectangular and triangular matrices. The only detail we have to do differently for triangular matrices is to directly return zero for their zero halves rather than accessing the corresponding container elements. We will encapsulate this detail in the function nonZeroRegion(), which takes the indices i and j and returns true if they address an element within the nonzero region of a matrix and false otherwise. We will have three different implementations of this function: one for rectangular matrices, one for lower triangular matrices, and one for upper triangular matrices. The implementation to be used in a given configuration of matrix components will be selected at compile time according to the value of the shape feature stored in Config. We will implement each variant of the function as a static function of a separate struct and use a metafunction to select the appropriate struct based on the current shape. Here is the implementation of nonZeroRegion for rectangular matrices (in Formats.h):

Module: Formats.h

```
namespace MatrixICCL{

struct RectNonZeroRegion
{ template<class M>
  static bool nonZeroRegion(const M* m, const M::Config::IndexType& i, const M::Config::IndexType& j)
  { return true;
  }
};
```

nonZeroRegion() takes a number of parameters which are not relevant here: we always return true since any of the elements of a rectangular matrix could be a nonzero element. This is different for a lower triangular matrix:

```
struct LowerTriangNonZeroRegion
{ template<class M>
  static bool nonZeroRegion(const M* m, const M::Config::IndexType& i, const M::Config::IndexType& j)
  { return i>=j;
  }
};
```

The first parameter of nonZeroRegion() is a pointer to the matrix format calling this function. We will see the point of call later. The only purpose of this parameter is to provide type information: we retrieve the index type from its Config. The nonzero region of an upper triangular matrix is just the negation of the above:

```
struct UpperTriangNonZeroRegion
{ template<class M>
  static bool nonZeroRegion(const M* m, const M::Config::IndexType& i, const M::Config::IndexType& j)
  {   return i<=j;
  }
};
```

The following is the metafunction selecting the appropriate implementation of nonZeroRegion() based on the matrix shape:

```
template<class MatrixType>
struct FORMAT_NON_ZERO_REGION
{ typedef MatrixType::Config Config;
  typedef Config::DSLFeatures DSLFeatures;
  typedef DSLFeatures::Shape Shape;

  typedef IF< EQUAL<Shape::id, Shape::lower_triang_id>::RET ||
        EQUAL<Shape::id, Shape::symm_id>::RET,
          LowerTriangNonZeroRegion,

      IF<EQUAL<Shape::id, Shape::upper_triang_id>::RET,
          UpperTriangNonZeroRegion,

      RectNonZeroRegion>::RET>::RET RET;
};
```

Thus, the metafunction uses a nested meta IF to select the appropriate implementation (we discussed metafunctions in Section 8.2). Now, we can implement ArrFormat as follows:

```
template<class Array>
class ArrFormat
{ public:
    typedef Array::Config Config;
    typedef Config::ElementType ElementType;
    typedef Config::IndexType IndexType;

  private:
    Array elements_;

  public:
    ArrFormat(const IndexType& r, const IndexType& c)
      : elements_(r, c)
    {}

    const IndexType& rows() const { return elements_.rows(); }
    const IndexType& cols() const { return elements_.cols(); }

    void setElement(const IndexType & i, const IndexType & j, const ElementType & v)
    { if (nonZeroRegion(i, j)) elements_.setElement(i, j, v);
      else assert(v == ElementType( 0 ));
    }

    ElementType getElement(const IndexType & i, const IndexType & j) const
    { return nonZeroRegion(i, j) ? elements_.getElement(i, j) : ElementType(0);
    }

    void initElements(const ElementType & v)
    { elements_.initElements(v);
    }

  protected:
    bool nonZeroRegion(const IndexType& i, const IndexType& j) const
    { return FORMAT_NON_ZERO_REGION<Config::MatrixType>::RET::nonZeroRegion(this,i, j);
    }
};
```

The last return demonstrates the call to nonZeroRegion(). The struct containing the appropriate function implementation is returned by the metafunction FORMAT_NON_ZERO_REGION<>. Since

we declared all implementations of nonZeroRegion() as static, inline functions of the structs, the C++ compiler should be able to inline this function to eliminate any overhead. This technique represents the static alternative to virtual functions.

LoTriangVecFormat stores the elements of a lower triangular matrix row-wise in a vector:

```
template<class Vector>
class LoTriangVecFormat
{ public:
    typedef Vector::Config Config;
    typedef Config::ElementType ElementType;
    typedef Config::IndexType IndexType;

  private:
    const order_; //number of rows and columns
    Vector elements_;

  public:
    LoTriangVecFormat(const IndexType& r,const IndexType& c)
      : order_(r), elements_(rows() * (rows() + 1) * 0.5)
    { assert(rows()==cols());
    }

    const IndexType& rows() const { return order_; }
    const IndexType& cols() const { return order_; }

    void setElement(const IndexType & i, const IndexType & j, const ElementType & v)
    { if (i >= j) elements_.setElement(getIndex(i, j), v);
      else assert(v == ElementType( 0 ));
    }

    ElementType getElement(const IndexType & i, const IndexType & j) const
    { return i >= j ? elements_.getElement(getIndex(i, j))
                    : ElementType(0);
    }

    void initElements(const ElementType & v)
    { elements_.initElements(v);
    }

  protected:
    IndexType getIndex(const IndexType& i, const IndexType& j) const
    { return (i + 1) * i * 0.5 + j;
    }
};
```

UpTriangVecFormat can be easily derived from LoTriangVecFormat. We only need to override setElement() and getElement() in order to swap the row and column index:

```
template<class Vector>
class UpTriangVecFormat : public LoTriangVecFormat<Vector>
{ public:
    UpTriangVecFormat(const IndexType & r,const IndexType & c)
      : LoTriangVecFormat(r, c)
    {}

    void setElement(const IndexType & i, const IndexType & j, const ElementType & v)
    { LoTriangVecFormat::setElement(j, i, v);
    }

    ElementType getElement(const IndexType & i, const IndexType & j) const
    { return LoTriangVecFormat::getElement(j, i);
    }
};
```

Finally, we need to implement Symm. Symm takes a lower triangular format as its parameter and turns it into a symmetric one. Symm is derived from its parameter and overrides setElement() and getElement():

```
template<class Format>
class Symm : public Format
{ public:
    typedef Format::Config Config;
    typedef Config::ElementType ElementType;
    typedef Config::IndexType IndexType;

    Symm(const IndexType& rows,const IndexType& cols)
      : Format(rows, cols)
    {}

    void setElement(const IndexType & i, const IndexType & j, const ElementType & v)
    { if( i >= j ) Format::setElement(i, j, v);
      else Format::setElement(j, i, v);
    }

    ElementType getElement(const IndexType & i, const IndexType & j) const
    { return ( i >= j ) ?
        Format::getElement(i, j) :
        Format::getElement(j, i);
    }
};

} //namespace MatrixICCL
```

10.3.1.4.3 Bounds Checking

Bounds checking is implemented by a wrapper similar to Symm discussed above. Here is the implementation (in BoundsChecker.h):

Module: BoundsChecker.h

```
namespace MatrixICCL{

template<class OptSymmMatrix>
class BoundsChecker : public BaseClass
{ public:
    typedef BaseClass::Config Config;
    typedef Config::ElementType ElementType;
    typedef Config::IndexType IndexType;

    BoundsChecker(const IndexType& r, const IndexType& c)
      : OptSymmMatrix(r, c)
    {}

    void setElement(const IndexType& i, const IndexType& j, const ElementType& v)
    { checkBounds(i, j);
      OptSymmMatrix::setElement(i, j, v);
    }

    ElementType getElement(const IndexType& i, const IndexType& j) const
    { checkBounds(i, j);
      return OptSymmMatrix::getElement(i, j);
    }

  protected:
    void checkBounds(const IndexType & i, const IndexType & j) const
    { if ( i < 0 || i >= rows() ||
           j < 0 || j >= cols()   )
        throw "subscript(s) out of bounds";
    }
};

} //namespace MatrixICCL
```

10.3.1.4.4 Matrix Wrapper

The top-level matrix component is Matrix<>. It defines a number of useful functions for all matrices: assignment operator for initializing a matrix using a comma-separated list of numbers, assignment operator for assigning binary expressions, assignment operator for assigning matrices to matrices, and a function for printing the contents of a matrix.

Module: TopWrapper.h

```
//declare BinaryExpression
template<class ExpressionType> class BinaryExpression;

namespace MatrixICCL{

template<class OptBoundsCheckedMatrix>
class Matrix : public OptBoundsCheckedMatrix
{ public:
    typedef OptBoundsCheckedMatrix::Config Config;

    typedef Config::IndexType IndexType;
    typedef Config::ElementType ElementType;
    typedef Config::CommaInitializer CommaInitializer;

    Matrix(IndexType rows= 0, IndexType cols= 0, ElementType InitElem = ElementType(0) )
      : OptBoundsCheckedMatrix(rows, cols)
    { initElements(InitElem);
    }

    //initialization by a comma-separated list of numbers
    CommaInitializer operator=(const ElementType& v)
    { return CommaInitializer(*this, v);
    }
```

The following assignment operator allows us to assign binary expressions to a matrix (we will discuss the class template BinaryExpression<> later):

```
    //assignment operator for binary expressions
    template <class Expr>
    Matrix& operator=(const BinaryExpression<Expr>& expr)
    { expr.assign(this);
      return *this;
    }
```

Finally, we have an assignment for assigning matrices to matrices. The implementation code depends on the shape of the source matrix. Thus, we use a similar technique as in the case of nonZeroRegion(): we select the appropriate implementation using a metafunction (the code for the metafunction and for the different assignment variants is given later):

```
    //matrix assignment
    template<class A>
    Matrix& operator=(const Matrix<A>& m)
    {
      MATRIX_ASSIGNMENT<A>::RET::assign(this, &m);
      return *this;
    }

    //assignment operators for other kinds of expressions
    //...

    //print matrix to ostream
    ostream& display(ostream& out) const
    { for( IndexType i = 0; i < rows(); ++i )
      { for( IndexType j = 0; j < cols(); ++j )
          out << getElement( i, j ) << "   ";
        out << endl;
      }
      return out;
    }
};
```

```
} //namespace MatrixICCL

//output operator for printing a matrix to a stream
template <class A>
ostream& operator<<(ostream& out, const Matrix<A>& m)
{
  return m.display(out);
}
```

10.3.1.5 Matrix Configuration Generator

The matrix configuration generator takes a matrix configuration description, e.g. matrix<double,rect<> >, and returns a matrix type with the properties specified in the configuration description. Given the above example configuration description, it generates the following matrix type:

Matrix<BoundsChecker<ArrFormat<Dyn2DCContainer<MATRIX_ASSEMBLE_COMPONENTS<...> > > > >

MATRIX_ASSEMBLE_COMPONENTS<> is the Generator parameter of Dyn2DCContainer we mentioned earlier. It has some parameters itself, but they are not relevant at this point (we indicated them by three dots). The only thing that matters here is that it contains Config, i.e. the configuration repository, as its member type and that Dyn2DCContainer<> can access it.

The generator performs three steps:

1. parsing the configuration description by reading out the nested DSL features;
2. assigning defaults to the unspecified DSL features;
3. assembling the matrix implementation components according to the DSL features.

These steps are shown in Figure 185. The processing steps and the corresponding metafunctions implementing them are enclosed in ellipses and the intermediate results are displayed in square boxes.

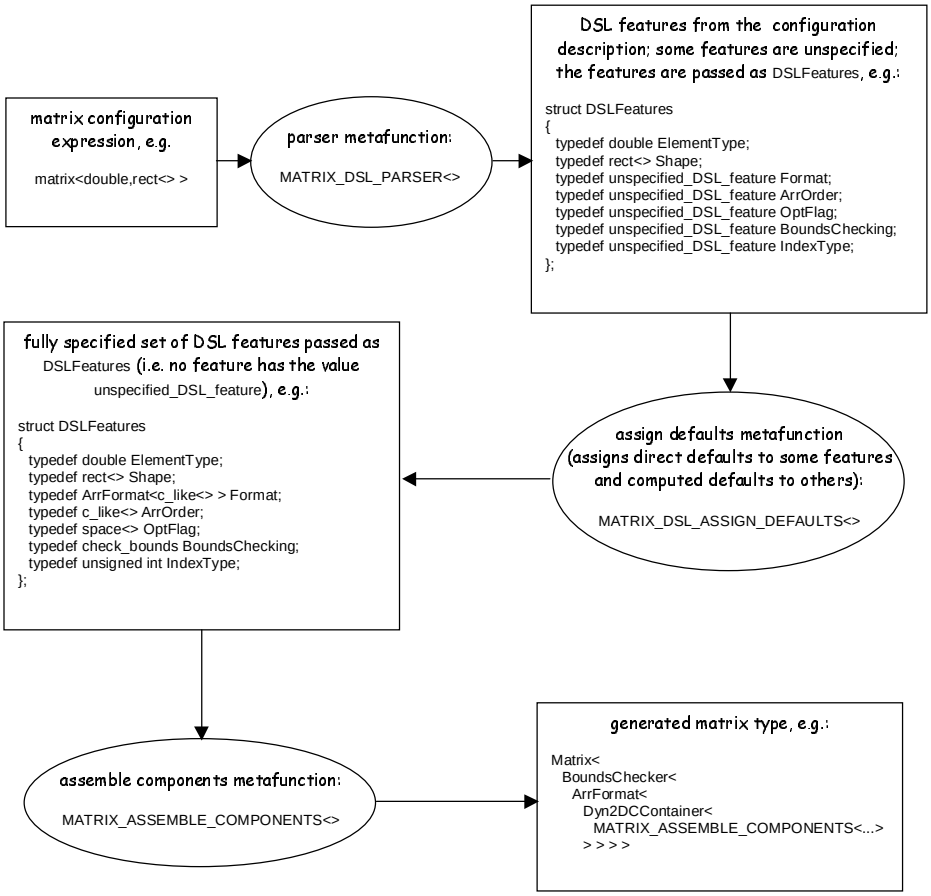

Figure 185 *Processing steps of the configuration generator*

Please note that the information about the matrix configuration passed between the processing steps is encoded by the DSLFeatures struct. This class contains a member type representing each of the DSL parameters (cf. Figure 182):

ElementType
Shape
Format
ArrayOrder
OptFlag
BoundsChecking
IndexType

This set of DSL parameters completely describes a matrix configuration. In other words, DSLFeatures represents the *type description record* of a matrix. We will use it later for selecting operation implementation and computing type records of expressions. We often refer to the DSLFeatures as the "flat" configuration class of a matrix.

Now we take a look at the implementation of the matrix generator. We implement it as a metafunction which takes two parameters: the matrix configuration description and a flag specifying what to do. The latter has the following purpose: We want to be able to pass not only a configuration DSL expression to the generator, but also DSLFeatures with some unspecified features and DSLFeatures with all features specified. In other words, if you take a look at Figure 185, we want to be able to enter the generation process just before any of the three processing steps. The reason for this is that we will sometimes already have DSLFeatures and just need to generate the corresponding matrix type. This is the case, for example, when we compute the type record of a matrix expression. We will see this later.

Thus, we first need to implement the flags: do_all, defaults_and_assemble, assemble_components. They have the following meaning:

- do_all: do parsing, assigning defaults, and component assembly; the generator expects a configuration DSL expression;
- defaults_and_assemble: do assigning defaults and component assembly; the generator expects DSLFeatures whose features do not have to be fully specified;
- assemble_components: do component assembly; the generator expects DSLFeatures whose features have to be fully specified;

We implement the flags as integer constants (in MatrixGenerator.h):

Module: MatrixGenerator.h

```
enum {
  do_all,
  defaults_and_assemble,
  assemble_components
};
```

Now, we can implement our configuration generator. As indicated in Figure 185, the generator delegates all the work to three other metafunctions, each of them defining one processing step:

```
MATRIX_DSL_PARSER<>
MATRIX_DSL_ASSIGN_DEFAULTS<>
MATRIX_ASSEMBLE_COMPONENTS<>
```

The generator returns (in its public section) the generated matrix type (as RET). The computation in the generator involves reading the "what to do" flag and calling the appropriate metafunctions:[167]

```
template< class InputDSL = matrix<>, int WhatToDo= do_all >
class MATRIX_GENERATOR
{
  // parse InputDSL (or dummy)
  typedef SWITCH< WhatToDo
            , CASE< assemble_components,    matrix<>        // dummy
            , CASE< defaults_and_assemble,  matrix<>        // dummy
            , DEFAULT<                      InputDSL
        > > > >::RET DSL_Description;
  typedef MATRIX_DSL_PARSER< DSL_Description >::RET ParsedDSL__;
```

Please note that we have to use the dummies since MATRIX_DSL_ASSIGN_DEFAULTS<> below will be "executed" in any case. We use the same calling pattern for the remaining two steps:

```
  // assign defaults to DSL (or to dummy)
  typedef SWITCH< WhatToDo
            , CASE< assemble_components,    ParsedDSL__  // dummy
            , CASE< defaults_and_assemble,  InputDSL
            , DEFAULT<                      ParsedDSL__
        > > > >::RET ParsedDSL_;
  typedef MATRIX_DSL_ASSIGN_DEFAULTS< ParsedDSL_ >::RET CompleteDSL__;

  // assemble components
  typedef SWITCH< WhatToDo
            , CASE< assemble_components,    InputDSL
            , CASE< defaults_and_assemble,  CompleteDSL__
            , DEFAULT<                      CompleteDSL__
        > > > >::RET CompleteDSL_;
  typedef MATRIX_ASSEMBLE_COMPONENTS< CompleteDSL_ > Result;
```

Finally, we have our public return:

```
public:
  typedef Result::RET     RET;
```

```
};
```

The following three sections describe the metafunctions implementing the three generation steps.

10.3.1.5.1 Configuration DSL Parser

The configuration DSL parser takes a matrix configuration DSL expression and produces DSLFeatures containing the values of the features explicitly specified in the configuration expression (the unspecified features have the value unspecified_DSL_feature; see Figure 185).

Before we show you the parser code, we need a small helper metafunction for testing whether a DLS feature is unspecified or not:

Module: DSLParser.h

```
using namespace MatrixDSL;

template<class TYPE> struct IsUnspecifiedDSLFeature {enum { RET=0 };};
template<> struct IsUnspecifiedDSLFeature<unspecified_DSL_feature>{enum { RET=1 };};
```

The parser retrieves all the features for the DSLFeatures class one at a time from the matrix configuration (i.e. its parameter DSLDescription) and returns the DSLFeatures:

```
template<class DSLDescription>
class MATRIX_DSL_PARSER
{ private:

    //ElementType
    typedef DSLDescription::elementType ElementType;

    //Shape
    typedef DSLDescription::shape Shape;

    //Format
    typedef DSLDescription::format Format;
```

The retrieval code for ArrOrder looks slightly different since this feature is nested only in the one value array<> of Format. Thus, we first have to check to see if the value of Format is array<...> and if this is not the case we simply return array<>, otherwise we return the actual value of Format. Then we use this intermediate result (i.e. ArrayFormat_) to read out the arr_order member. The intermediate result (i.e. ArrayFormat_) always contains arr_order, but its value is only relevant if the value of Feature is array<...>. We have to go through all this trouble since both types passed to a meta IF are actually built and we have to make sure that _ArrayOrder has arr_order as its member type in all cases. Here is the code:

```
    //ArrOrder
    typedef IF<EQUAL<Format::id, Format::array_id>::RET,
          Format,
          array<> >::RET ArrayFormat_;
    typedef IF<EQUAL<Format::id, Format::array_id>::RET,
          ArrayFormat_::arr_order,
          unspecified_DSL_feature>::RET ArrOrder;
```

The remaining parameters are simple to retrieve:

```
    //OptFlag
    typedef DSLDescription::optFlag OptFlag;

    //BoundsChecking
    typedef DSLDescription::boundsChecking BoundsChecking;

    //IndexType
    typedef DSLDescription::indexType IndexType;
```

Finally, we define DSLFeatures with all the DSL parameters as its member types and return it:

```
  public:
    struct DSLFeatures
    {
      typedef ElementType ElementType;
      typedef Shape Shape;
      typedef Format Format;
      typedef ArrOrder ArrOrder;
      typedef OptFlag OptFlag;
      typedef BoundsChecking BoundsChecking;
      typedef IndexType IndexType;
    };

    typedef DSLFeatures RET;
};
```

10.3.1.5.2 Assigning Defaults to DSL Features

The second generation step is to assign default values to the unspecified features in DSLFeatures returned by the parser (see Figure 185). Some features are assigned direct default values and some computed default values. Thus, we first need to specify the direct and the computed defaults for our DSL.

ElementType:	double
Shape:	rect
ArrayOrder:	cLike
OptFlag:	space
BoundsChecking:	checkBounds
IndexType:	unsigned int

Table 101 *Direct feature defaults for the demo matrix package*

The direct defaults are listed in Table 101. As already discussed, some choices might be a matter of taste, but we have to make them in any case (see discussion in Section 10.2.3).

The only computed default is Format and the computation formula is given in Table 102. According to this table, Format depends on Shape and OptFlag. We only use vector for triangular and symmetric matrices if the optimization flag is set to space.

Shape	OptFlag	Format
rect	*	array
lowerTriang upperTriang symm	speed	array
	space	vector

Table 102 *Computing default value for* Format

There is one more detail we have to specify: It is illegal to set Shape to rect and Format to vector at the same time. This combination does not seem to be useful and we choose to forbid it explicitly. This is specified in Table 103.

Shape	Format
rect	vector

Table 103 *Illegal feature combination*

Given the above specifications, we can move to the C++ implementation now. First, we implement the table with the direct feature defaults (i.e. Table 101) in a struct. This way we have all direct defaults in one place, which is desirable for maintenance reasons (in DSLAssignDefaults.h):

Module: DSLAssignDefaults.h

```
using namespace MatrixDSL;

//DSLFeatureDefaults implements Table 101 (i.e. direct feature defaults)
```

```
struct DSLFeatureDefaults
{ typedef double              ElementType;
  typedef rect<>              Shape;
  typedef c_like<>            ArrOrder;
  typedef space<>             OptFlag;
  typedef check_bounds<>      BoundsChecking;
  typedef unsigned int        IndexType;
};
```

The metafunction for assigning feature defaults does more than just assigning defaults: it also checks to make sure that the specified feature values are correct and that the feature combinations are correct. Thus, for example, if we specified the shape of a matrix as speed<>, this would be the place to catch this error.

The implementation of error checking is a bit tricky. A major deficiency of template metaprogramming is that we do not have any means to report a string to the programmer during compilation. In our code, we use the following partial solution to this problem: If we want to report an error, we access the nonexistent member SOME_MEMBER of some type SOME_TYPE which, of course, causes the compiler to issue a compilation error that says something like: "'SOME_MEMBER' is not a member of 'SOME_TYPE'". Now, the idea is to use the name of the kind of error we want to report as SOME_TYPE and to encode the error text in the name of SOME_MEMBER.

Here is the implementation of the "SOME_TYPE", which, in our case, we call DSL_FEATURE_ERROR:

```
struct DSL_FEATURE_ERROR {};
```

DSL_FEATURE_ERROR is usually returned by a meta IF. Thus, we also need some type to return if there is no error:

```
struct nil {}; // nil is just some other type

struct DSL_FEATURE_OK
{
  typedef nil WRONG_SHAPE;
  typedef nil WRONG_FORMAT_OR_FORMAT_SHAPE_COMBINATION;
  typedef nil WRONG_ARR_ORDER;
  typedef nil WRONG_OPT_FLAG;
  typedef nil WRONG_BOUNDS_CHECKING;
};
```

As you see, DSL_FEATURE_OK encodes error strings as member type names.

Now, for each DSL parameter, we implement a checking metafunction, which checks if the value of the parameter is one of the valid values according to the DSL grammar (see Figure 182):

```
template<class Shape>
struct CheckShape
{
  typedef IF< EQUAL<Shape::id, Shape::rect_id>::RET ||
        EQUAL<Shape::id, Shape::lower_triang_id>::RET ||
        EQUAL<Shape::id, Shape::upper_triang_id>::RET ||
        EQUAL<Shape::id, Shape::symm_id>::RET,
          DSL_FEATURE_OK,
          DSL_FEATURE_ERROR>::RET::WRONG_SHAPE RET;
};
```

Thus, if Shape is neither rect<>, nor lower_triang<>, nor upper_triang<>, nor symm<>, the meta IF returns DSL_FEATURE_ERROR and trying to access WRONG_SHAPE of DSL_FEATURE_ERROR results in a compilation error. In the other case, we return DSL_FEATURE_OK, which, as we already saw, defines WRONG_SHAPE as its member.

In practice, our error reporting approach looks as follows: If we try to compile the following line:

```
typedef MATRIX_GENERATOR<  matrix<  double, speed<>> >::RET MyMatrixType;
```

the C++ compiler (in our case VC++5.0) will issue the following error:

```
error C2039: '  WRONG_SHAPE'   : is not a member of '  DSL_FEATURE_ERROR'
```

This error indicates that the second parameter to MATRIX_GENERATOR<> is not a valid shape value. This is a useful hint. Unfortunately, it does not tell us the source line where we used the wrong parameter value but rather the line where the erroneous member access occurred.

The checking of Format is a bit more complex since we need to implement Table 103. This table stated that we do not want to allow a rectangular matrix to be stored in a vector. Thus, the following metafunction takes Format and Shape as its parameters:

```
template<class Format, class Shape>
struct CheckFormatAndFormatShapeCombination
{
  typedef IF<(EQUAL<Shape::id, Shape::rect_id>::RET &&
          EQUAL<Format::id, Format::array_id>::RET) ||

         ((EQUAL<Shape::id, Shape::lower_triang_id>::RET ||
          EQUAL<Shape::id, Shape::upper_triang_id>::RET ||
          EQUAL<Shape::id, Shape::symm_id>::RET) &&
            (EQUAL<Format::id, Format::vector_id>::RET ||
             EQUAL<Format::id, Format::array_id>::RET)),

          DSL_FEATURE_OK,
          DSL_FEATURE_ERROR>::RET::WRONG_FORMAT_OR_FORMAT_SHAPE_COMBINATION RET;
};
```

Here are the checking metafunctions for the remaining three DSL parameters:

```
template<class ArrOrder>
struct CheckArrOrder
{
  typedef IF< EQUAL<ArrOrder::id, ArrOrder::c_like_id>::RET ||
          EQUAL<ArrOrder::id, ArrOrder::fortran_like_id>::RET,
            DSL_FEATURE_OK,
            DSL_FEATURE_ERROR>::RET::WRONG_ARR_ORDER RET;
};


template<class OptFlag>
struct CheckOptFlag
{
  typedef IF< EQUAL<OptFlag::id, OptFlag::speed_id>::RET ||
          EQUAL<OptFlag::id, OptFlag::space_id>::RET,
            DSL_FEATURE_OK,
            DSL_FEATURE_ERROR>::RET::WRONG_OPT_FLAG RET;
};


template<class BoundsChecking>
struct CheckBoundsChecking
{
  typedef IF< EQUAL<BoundsChecking::id, BoundsChecking::check_bounds_id>::RET ||
          EQUAL<BoundsChecking::id, BoundsChecking::no_bounds_checking_id>::RET,
            DSL_FEATURE_OK,
            DSL_FEATURE_ERROR>::RET::WRONG_BOUNDS_CHECKING RET;
};
```

Please note that we do not provide checking for element type and index type. The reason is that we do not want to unnecessarily limit the number of types that can be used in their place. This is particularly true of element type since we also would like to be able to create matrices of some user-defined types. If we use matrix expressions, the element type will also have to implement

numeric operators such as +, *, -, +=, etc. If it does not, the C++ compiler will report an error from within the matrix operator code.

Finally, we are ready to implement our metafunction for assigning defaults:

```
template<class ParsedDSLDescription>
class MATRIX_DSL_ASSIGN_DEFAULTS
{ private:
    //define a short alias for the parameter
    typedef ParsedDSLDescription ParsedDSL;

    //ElementType
    typedef IF<IsUnspecifiedDSLFeature<ParsedDSL::ElementType>::RET,
          DSLFeatureDefaults::ElementType,
          ParsedDSL::ElementType>::RET ElementType;
```

If the element type is unspecified, the code above assigns it the direct default from Table 101. We do the same for IndexType, Shape, ArrOrder, OptFlag, and BoundsChecking:

```
    //IndexType
    typedef IF<IsUnspecifiedDSLFeature<ParsedDSL::IndexType>::RET,
          DSLFeatureDefaults::IndexType,
          ParsedDSL::IndexType>::RET IndexType;

    //Shape
    typedef IF<IsUnspecifiedDSLFeature<ParsedDSL::Shape>::RET,
          DSLFeatureDefaults::Shape,
          ParsedDSL::Shape>::RET Shape;
    typedef CheckShape<Shape>::RET check_shape_;
```

The last typedef calls the checking metafunction for Shape. We always call the checking metafunction after assigning the default value:

```
    //ArrOrder
    typedef IF<IsUnspecifiedDSLFeature<ParsedDSL::ArrOrder>::RET,
          DSLFeatureDefaults::ArrOrder,
          ParsedDSL::ArrOrder>::RET ArrOrder;
    typedef CheckArrOrder<ArrOrder>::RET check_arr_order_;

    //OptFlag
    typedef IF<IsUnspecifiedDSLFeature<ParsedDSL::OptFlag>::RET,
          DSLFeatureDefaults::OptFlag,
          ParsedDSL::OptFlag>::RET OptFlag;
    typedef CheckOptFlag<OptFlag>::RET check_opt_flag_;

    //BoundsChecking
    typedef IF<IsUnspecifiedDSLFeature<ParsedDSL::BoundsChecking>::RET,
          DSLFeatureDefaults::BoundsChecking,
          ParsedDSL::BoundsChecking>::RET BoundsChecking;
    typedef CheckBoundsChecking<BoundsChecking>::RET check_bounds_checking_;
```

Format is a special case since it does not have a direct default value. Its default value is determined based on Shape and OptFlag, as specified in Table 102:

```
    //Format
    typedef
      IF<  (EQUAL<Shape::id, Shape::lower_triang_id>::RET ||
         EQUAL<Shape::id, Shape::upper_triang_id>::RET ||
         EQUAL<Shape::id, Shape::symm_id>::RET) &&
          EQUAL<OptFlag::id, OptFlag::space_id>::RET,

          vector<>,
          array<> >::RET ComputedFormat_;

    typedef IF<IsUnspecifiedDSLFeature<ParsedDSL::Format>::RET,
      ComputedFormat_,
```

```
    ParsedDSL::Format>::RET Format;
```

Next, we need to check the format-shape combination (cf. Table 103):

```
  typedef CheckFormatAndFormatShapeCombination<Format, Shape>::RET
    check_format_and_format_shape_combination_;
```

Finally, we return the DSLFeatures containing all DSL parameters:

```
public:
  struct DSLFeatures
  {
      typedef ElementType ElementType;
      typedef Shape Shape;
      typedef Format Format;
      typedef ArrOrder ArrOrder;
      typedef OptFlag OptFlag;
      typedef BoundsChecking BoundsChecking;
      typedef IndexType IndexType;
  };

  typedef DSLFeatures RET;
};
```

10.3.1.5.3 Matrix Component Assembler

The final step in the matrix type generation is assembling the matrix implementation components according to the matrix type record (i.e. DSLFeatures) produced in the earlier stages (see Figure 185).

First, we need to specify how to compute the ICCL parameters from the DSL parameters. According to the ICCL grammar in Figure 184, we have the following ICCL parameters with alternative values:

Array
Format
OptSymmetricMatrix
OptBoundsCheckedMatrix

Their computation from the DSL parameters is specified in Table 104 through Table 107.

ArrOrder	Array
cLike	Dyn2DCContainer
fortranLike	Dyn2DFContainer

Table 104 *Table for computing the ICCL parameter* Array

Shape	Format (DSL)	Format (ICCL)
*	array	ArrFormat
lowerTriangular symmetric	vector	LoTriangVecFormat
upperTriangular	vector	UpTriangVecFormat

Table 105 *Table for computing the ICCL parameter* Format

Shape	OptSymmetricMatrix
symm	Symm
*	= Format (ICCL)

Table 106 *Table for computing the ICCL parameter* OptSymmetricMatrix

BoundsChecking	OptBoundsCheckedMatrix
checkBounds	BoundsChecker
noBoundsChecking	= OptSymmetricMatrix

Table 107 *Table for computing the ICCL parameter* OptBoundsCheckedMatrix

According to the ICCL grammar, the value of Vector is Dyn1DContainer[Config]. Finally, the types provided by the configuration repository (i.e. Config, see Figure 183) are determined using Table 108 and Table 109.

ElementType (ICCL)	= ElementType (DSL)
IndexType (ICCL)	= IndexType (DSL)

Table 108 *Table for computing the ICCL parameters ElementType and IndexType*

ArrOrder	CommaInitializer
cLike	DenseCCommaInitializer
fortranLike	DenseFCommaInitializer

Table 109 *Table for computing the ICCL parameter* CommaInitializer

The metafunction for assembling components is implemented as follows (in AssembleComponents.h):

Module: AssembleComponents.h

```
using namespace MatrixDSL;
using namespace MatrixICCL;

template<class CompleteDSLDescription>
class MATRIX_ASSEMBLE_COMPONENTS
{ private:
    //introduce the alias Generator for itself
    typedef MATRIX_ASSEMBLE_COMPONENTS<CompleteDSLDescription> Generator;
    //introduce short alias for the parameter
    typedef CompleteDSLDescription DSLFeatures;
```

Now, each of the following typedefs implements one ICCL parameter (we cite the corresponding specification tables in the comments):

```
    //ElementType (see Table 108)
    typedef DSLFeatures::ElementType ElementType;

    //IndexType (see Table 108)
    typedef DSLFeatures::IndexType IndexType;

    //Vector (see Figure 184)
    typedef Dyn1DContainer<Generator> Vector;

    //Array (see Table 104)
```

```
typedef IF<EQUAL<DSLFeatures::ArrOrder::id, DSLFeatures::ArrOrder::c_like_id>::RET,
      Dyn2DCContainer<Generator>,
      Dyn2DFContainer<Generator> >::RET Array;
```

Please note that we passed Generator to the basic containers in the above typedefs for Vector and Array. Here are the remaining ICCL parameters:

```
//Format (see Table 105)
typedef
  IF<  (EQUAL<DSLFeatures::Shape::id, DSLFeatures::Shape::lower_triang_id>::RET ||
       EQUAL<DSLFeatures::Shape::id, DSLFeatures::Shape::symm_id>::RET) &&
       EQUAL<DSLFeatures::Shape::id, DSLFeatures::Format::vector_id>::RET,
         LoTriangVecFormat<Vector>,

  IF<  EQUAL<DSLFeatures::Shape::id, DSLFeatures::Shape::upper_triang_id>::RET &&
       EQUAL<DSLFeatures::Shape::id, DSLFeatures::Format::vector_id>::RET,
         UpTriangVecFormat<Vector>,

  ArrFormat<Array> >::RET>::RET Format;

//OptSymmetricMatrix (see Table 106)
typedef IF<EQUAL<DSLFeatures::Shape::id, DSLFeatures::Shape::symm_id>::RET,
    Symm<Format>,
    Format>::RET OptSymmetricMatrix;

//OptBoundsCheckedMatrix (see Table 107)
typedef IF<EQUAL<DSLFeatures::BoundsChecking::id,
                 DSLFeatures::BoundsChecking::check_bounds_id>::RET,
    BoundsChecker<OptSymmetricMatrix>,
    OptSymmetricMatrix>::RET OptBoundsCheckedMatrix;

//MatrixType;
typedef Matrix<OptBoundsCheckedMatrix> MatrixType;

//CommaInitializer (see Table 109)
typedef

  IF<EQUAL<DSLFeatures::ArrOrder::id, DSLFeatures::ArrOrder::c_like_id>::RET,
    DenseCCommaInitializer<Generator>,
    DenseFCommaInitializer<Generator> >::RET CommaInitializer;
```

Finally, we return the Config and the generated matrix type. Since we passed Generator to the basic containers, they have access to Config. They also pass it to the components of the upper layers.

```
public:
  struct Config
  {
    //DSL features
    typedef DSLFeatures DSLFeatures;

    //ICCL features
    typedef ElementType ElementType;
    typedef IndexType IndexType;

    //MatrixType
    typedef MatrixType MatrixType;

    typedef CommaInitializer CommaInitializer;
  };

  typedef MatrixType RET; //here is our generated matrix type!!!
};
```

10.3.1.6 A More Intentional Alternative to Nested IFs

As you saw in the previous two sections, the implementation of assigning defaults and assembling components involves implementing the dependency tables for computing defaults and for computing ICCL parameters. The technique we used for implementing those tables were nested

meta IFs. This technique was satisfactory for our demo matrix component, but the tables for the full matrix component specified in Section 10.2 are much bigger.

We demonstrate our point using the still relatively simple table for computing the ICCL parameter Array from Section 10.2.5.1 (i.e. Table 59):

Malloc	ArrOrder	Array
fix	cLike	Fix2DCContainer
	fortranLike	Fix2DFContainer
dyn	cLike	Dyn2DCContainer
	fortranLike	Dyn2DFContainer

The nested-meta-IF implementation of this table looks like this:

```
//Arr
    typedef IF<EQUAL<DSLFeatures::Malloc::id, DSLFeatures::Malloc::fix_id>::RET,
          IF<EQUAL<DSLFeatures::ArrOrder::id, DSLFeatures::ArrOrder::c_like_id>::RET,
            Fix2DCContainer<Size, Generator>,
          IF<EQUAL<DSLFeatures::ArrOrder::id, DSLFeatures::ArrOrder::fortran_like_id>::RET,
            Fix2DFContainer<Size, Generator>,
          invalid_ICCL_feature>::RET>::RET,

        IF<EQUAL<DSLFeatures::Malloc::id, DSLFeatures::Malloc::dyn_id>::RET,
          IF<EQUAL<DSLFeatures::ArrOrder::id, DSLFeatures::ArrOrder::c_like_id>::RET,
            Dyn2DCContainer<Generator>,
          IF<EQUAL<DSLFeatures::ArrOrder::id, DSLFeatures::ArrOrder::fortran_like_id>::RET,
            Dyn2DFContainer<Generator>,
          invalid_ICCL_feature>::RET>::RET,

        invalid_ICCL_feature>::RET>::RET Arr;
```

This is certainly not very readable. We can improve the situation by providing a metafunction allowing us to encode the tables more directly. This metafunction takes two parameters:

```
EVAL_DEPENDENCY_TABLE<HeadRow, TableBody>
```

HeadRow represents the head row of the dependency table (i.e. the first, gray row) and TableBody represents the remaining rows. Both parameters are implemented as lists (by nesting class templates). Here is an example:

```
typedef    EVAL_DEPENDENCY_TABLE

           <        CELL< 1,   CELL< 2                    > >

           , ROW< CELL< 4,   CELL< 3,   RET< Foo1  > > >
           , ROW< CELL< 1,   CELL< 5,   RET< Foo2  > > >
           , ROW< CELL< 1,   CELL< 2,   RET< Foo3  > > >
           , ROW< CELL< 2,   CELL< 3,   RET< Foo4  > > >
            > > > >  >::RET result;                          // result is Foo3
```

EVAL_DEPENDENCY_TABLE<> does the following: It looks for a row in TableBody whose cells match the cells of TableHead starting with the first row in TableBody. If it finds a matching row, it returns the corresponding return type (i.e. the type wrapped in RET<>; you can nest RETs in order to return more than one type, e.g. RET<Foo1, RET<Foo2> >). If it does not find a matching row, it reports an error. You can use anyValue as an asterisk (i.e. a value that matches anything).

Using this metafunction, we can rewrite our table as follows:

```
typedef DSLFeatures::Malloc     Malloc_;
typedef DSLFeatures::ArrOrder ArrOrder_;
typedef invalid_ICCL_feature     invalid;

enum {
  mallocID      = Malloc_::id,
  dyn           = Malloc_::dyn_id,
  fix           = Malloc_::fix_id,

  arrOrdID      = ArrOrder_::id,
  cLike         = ArrOrder_::c_like_id,
  fortranLike   = ArrOrder_::fortran_like_id
}

// Arr
  typedef EVAL_DEPENDENCY_TABLE                          // tables 16, 20
  //*************************************************************************************
  <          CELL< mallocID,   CELL< arrOrdID                                              > >

  , ROW< CELL< fix,           CELL< cLike,        RET< Fix2DCContainer< Size, Generator > > > >
  , ROW< CELL< fix,           CELL< fortranL,     RET< Fix2DFContainer< Size, Generator > > > >
  , ROW< CELL< dyn,           CELL< cLike,        RET< Dyn2DCContainer< Generator >      > > >
  , ROW< CELL< dyn,           CELL< fortranL,     RET< Dyn2DFContainer< Generator >      > > >
  , ROW< CELL< anyValue,      CELL< anyValue,     RET< invalid                           > > >
  //*************************************************************************************
  > > > > > >::RET Arr;
```

The implementation of this function is given in Section 10.5.

Unfortunately, compared to the nested-IFs solution, the use of EVAL_DEPENDENCY_TABLE<> increases compilation times of the matrix component by an order of magnitude (EVAL_DEPENDENCY_TABLE<> uses recursion).

10.3.1.7 Matrix Operations

We implement matrix operations such as multiplication and addition using the technique of *expression templates* [Vel95]. We already explained the basic idea behind this implementation in Sections 10.2.6.1 and 10.2.6.2. We give you the C++ code in six parts:

1. *matrix operator templates*: operator templates implementing the operators + and * for matrices;

2. *matrix expression templates*: class templates for representing addition and multiplication expressions;

3. *matrix cache*: cache for implementing matrix multiplication (see the lazy-with-cache variant in Section 10.2.6.1);

4. getElement() *for expressions*: getElement() returns one element of a matrix expression; there will be different implementations for different shape combinations of the argument matrices;

5. *metafunctions for computing result types of expressions*: metafunctions for computing the matrix type of an expression, i.e. the type which is appropriate for storing the result of evaluating the expression;

6. *assignment functions for assigning expressions and matrices to matrices*: there will be different implementation of the assignment functions for different shapes of the source matrix (or expression).

We only consider the implementation of matrix addition and multiplication since matrix subtraction is similar to matrix addition.

10.3.1.7.1 Matrix Operator Templates

The main idea behind expression templates is the following: if you add two matrices, e.g. A+B, you do not return the result matrix, but an object representing the addition expression instead. If you have a more complicated expression, e.g. (A+B)*(C+D), you return a nested expression object (see Figure 177). This is done as follows: You first execute the two plus operators. They both return a matrix addition expressions. Finally, you execute the multiplication operator, which returns a multiplication expression pointing to the other two addition expressions as its argument objects. An expression object can be accessed using getElement() – just as any matrix.

Thus, in any case, you end up with an expression object. Since we implement the operators + and * using overloaded operator templates, we will know the complete type of a complex expression at compile time. The expression type describes the structure of the expression and we can pass it to metafunctions analyzing the expression structure and generating optimized code for the methods of the expression. However, the matrix expression optimization we implement later will involve the inspection of two expression argument types at a time rather than analyzing whole expression structures (the latter could be necessary for other kinds of optimizations). Depending on the shape combination of the arguments, we will select different implementations of the getElement() (i.e. the function for computing expression elements).

According to the approach outlined above, we need the following operator implementations:

- + for two matrices, e.g. A+B
- + for a matrix and an addition expression, e.g. A+(B+C)
- + for an addition expression and a matrix, e.g. (A+B)+C
- + for two addition expressions, e.g. (A+B)+(C+D)

Furthermore, we would need a similar set of four implementations of * and implementations for all the combinations of addition and multiplication expressions, e.g. (A+B)*C, (A+B)*(C+D), etc.

We can avoid this combinatorial explosion by wrapping the multiplication and the addition expressions into binary expressions. In this case, we only need four implementations of + and four implementations of *. The C++ implementation looks as follows (in MatrixOperTemplates.h):

Module: MatrixOperTemplates.h

```
/*** Addition **/

//Matrix + Matrix
template <class M1, class M2>
inline BinaryExpression<AdditionExpression<Matrix<M1>, Matrix<M2> > >
operator+(const Matrix<M1>& m1, const Matrix<M2>& m2)
{ return BinaryExpression<AdditionExpression<Matrix<M1>, Matrix<M2> > >(m1, m2);
}

//Expression + Matrix
template <class Expr, class M>
inline BinaryExpression<AdditionExpression<BinaryExpression<Expr>, Matrix<M> > >
operator+(const BinaryExpression<Expr>& expr, const Matrix<M>& m)
{ return BinaryExpression<AdditionExpression<BinaryExpression<Expr>, Matrix<M> > >(expr, m);
}

//Matrix + Expression
template <class M, class Expr>
inline BinaryExpression<AdditionExpression<Matrix<M>, BinaryExpression<Expr> > >
```

```
operator+(const Matrix<M>& m, const BinaryExpression<Expr>& expr)
{ return BinaryExpression<AdditionExpression<Matrix<M>, BinaryExpression<Expr> > >(m, expr);
}

//Expression + Expression
template <class Expr1, class Expr2>
inline BinaryExpression<AdditionExpression<BinaryExpression<Expr1>, BinaryExpression<Expr2> > >
operator+(const BinaryExpression<Expr1>& expr1, const BinaryExpression<Expr2>& expr2)
{ return BinaryExpression<AdditionExpression<  BinaryExpression<Expr1>,
                                               BinaryExpression<Expr2> > >(expr1, expr2);
}


/*** Multiplication **/

//Matrix * Matrix
template <class M1, class M2>
inline BinaryExpression<MultiplicationExpression<Matrix<M1>, Matrix<M2> > >
operator*(const Matrix<M1>& m1, const Matrix<M2>& m2)
{ return BinaryExpression<MultiplicationExpression<Matrix<M1>, Matrix<M2> > >(m1, m2);
}

//Expression * Matrix
template <class Expr, class M>
inline BinaryExpression<MultiplicationExpression<BinaryExpression<Expr>, Matrix<M> > >
operator*(const BinaryExpression<Expr>& expr, const Matrix<M>& m)
{ return BinaryExpression<MultiplicationExpression<BinaryExpression<Expr>, Matrix<M> > >(expr, m);
}

//Matrix * Expression
template <class M, class Expr>
inline BinaryExpression<MultiplicationExpression<Matrix<M>, BinaryExpression<Expr> > >
operator*(const Matrix<M>& m, const BinaryExpression<Expr>& expr)
{ return BinaryExpression<MultiplicationExpression<Matrix<M>, BinaryExpression<Expr> > >(m, expr);
}

//Expression * Expression
template <class Expr1, class Expr2>
inline BinaryExpression<MultiplicationExpression<BinaryExpression<Expr1>, BinaryExpression<Expr2> > >
operator*(const BinaryExpression<Expr1>& expr1, const BinaryExpression<Expr2>& expr2)
{ return BinaryExpression<MultiplicationExpression<  BinaryExpression<Expr1>,
                                                     BinaryExpression<Expr2> > >(expr1, expr2);
}
```

The operator templates can be thought of as a parsing facility. For example, given the above operator templates and the two matrices RectMatrix1 and RectMatrix2 of type

```
Matrix<BoundsChecker<ArrFormat<Dyn2DCContainer<MATRIX_ASSEMBLE_COMPONENTS<...>>>>>
```

the C++ compiler derives for the following expression

```
(RectMatrix1 + RectMatrix2)*(RectMatrix1 + RectMatrix2)
```

the following type:

```
BinaryExpression<
  MultiplicationExpression<
    BinaryExpression<
      AdditionExpression<
        Matrix<BoundsChecker<ArrFormat<Dyn2DCContainer<MATRIX_ASSEMBLE_COMPONENTS<...>>>>>,
        Matrix<BoundsChecker<ArrFormat<Dyn2DCContainer<MATRIX_ASSEMBLE_COMPONENTS<...>>>>>
      >
    >,
    BinaryExpression<
      AdditionExpression<
        Matrix<BoundsChecker<ArrFormat<Dyn2DCContainer<MATRIX_ASSEMBLE_COMPONENTS<...>>>>>,
        Matrix<BoundsChecker<ArrFormat<Dyn2DCContainer<MATRIX_ASSEMBLE_COMPONENTS<...>>>>>
      >
    >
  >
>
```

10.3.1.7.2 Matrix Expression Templates

The class template AdditionExpression<> represents the addition of two arguments, MultiplicationExpression<> represents the multiplication of two operands, and BinaryExpression<> is used to wrap the previous two to make them look alike.

We start with the implementation of AdditionExpression<> (in MatrixExprTemplates.h):

Module: MatrixExprTemplates.h

```
template<class A, class B>
class AdditionExpression
{ public:
    typedef A LeftType;
    typedef B RightType;
```

Any of LeftType and RightType can be either a matrix type or a binary expression type. Next, we need to compute the result matrix type of the addition expression, i.e. matrix type which would be appropriate for storing the result of the evaluation of this expression. We compute the result type using the metafunction ADD_RESULT_TYPE<>, which we discuss later. We publish the configuration repository of the result type in the member Config of AdditionExpression<>. Thus, an addition expression has a Config describing its matrix type – just as any matrix type does. Indeed, ADD_RESULT_TYPE<> uses the Config of the operands in order to compute the Config of the result:

```
    typedef ADD_RESULT_TYPE<LeftType, RightType>::RET::Config Config;
```

Next, we read out the element type and the index type for this expression from the resulting Config:

```
    typedef Config::ElementType ElementType;
    typedef Config::IndexType IndexType;
```

The addition expression needs two variables to point to its operands:

```
  private:
    const LeftType& _left;
    const RightType& _right;

  protected:
    const IndexType rows_, cols_;
```

The constructor initializes the expression variables and checks if the dimensions of the operands are compatible:

```
  public:
    AdditionExpression(const LeftType& m1, const RightType& m2)
      : _left(m1), _right(m2),
      rows_(m1.rows()), cols_(m1.cols())
    { if (m1.cols() != m2.cols() || m1.rows() != m2.rows())
```

```
        throw "argument matrices are incompatible";
    }
```

The addition expression defines a getElement() method for accessing its matrix elements as any matrix does. However, in this case, each element is computed from the operands rather than stored directly. We use a metafunction to select the most appropriate implementation of getElement() based on the shape of the operands (we will discuss this function later):

```
    ElementType getElement( const IndexType & i, const IndexType & j ) const
    { return MATRIX_ADD_GET_ELEMENT<LeftType, RightType>::RET::getElement(i, j, this, _left, _right);
    }

    IndexType rows() const {return rows_;}
    IndexType cols() const {return cols_;}
};
```

The beginning of MultiplicationExpression looks similarly to AdditionExpression, except that we additionally read out the left and the right matrix type to be used for deriving operand caches later:

```
template<class A, class B>
class MultiplicationExpression
{ public:
    typedef A LeftType;
    typedef B RightType;
    typedef LeftType::Config::MatrixType LeftMatrixType;
    typedef RightType::Config::MatrixType RightMatrixType;

    typedef MULTIPLY_RESULT_TYPE<LeftType, RightType>::RET::Config Config;

    typedef Config::ElementType ElementType;
    typedef Config::IndexType IndexType;
```

As we explained in Section 10.2.6.1, in the case of matrix multiplication, we need to use caches for those operands which are expressions themselves. The reason was that matrix multiplication accesses each element of the operands more than one time and by using a cache we avoid the recalculation of the elements of the operand expression on each access. The type of the caches is computed from the corresponding operand types by a metafunction, which we discuss later. Finally, we provide variables for keeping track of the operands and the caches (if any):

```
  private:
    typedef CACHE_MATRIX_TYPE<LeftMatrixType>::RET LeftCacheMatrixType;
    typedef CACHE_MATRIX_TYPE<RightMatrixType>::RET RightCacheMatrixType;

    const LeftType& _left;
    const RightType& _right;

    LeftCacheMatrixType* _left_cache_matrix;
    RightCacheMatrixType* _right_cache_matrix;

  protected:
    const IndexType rows_, cols_;
```

The multiplication expression needs four constructors, each of them for one of the following combinations:

- both operands are simple matrices;
- left operand is a matrix and right operand is an expression;
- right operand is an expression and left operand is a matrix;
- both operands are expressions.

We start with two matrices. In this case we do not need any caches:

```
public:
  template<class M1, class M2>
  MultiplicationExpression(const Matrix<M1>& m1, const Matrix<M2>& m2)
    : _left(m1), _right(m2),
    _left_cache_matrix(NULL), _right_cache_matrix(NULL),
    rows_(m1.rows()), cols_(m2.cols())
  { ParameterCheck(m1, m2);
  }
```

The following two constructors have to create a cache for one of the two operands:

```
  template<class Expr, class M2>
  MultiplicationExpression(const BinaryExpression<Expr>& expr, const Matrix<M2>& m)
    : _left(expr),  _right(m),
    _right_cache_matrix(NULL),
    rows_(expr.rows()), cols_(m.cols())
  { ParameterCheck(expr, m);
    _left_cache_matrix = new LeftCacheMatrixType(expr.rows(), expr.cols());
  }

  template<class M, class Expr>
  MultiplicationExpression(const Matrix<M>& m, const BinaryExpression<Expr>& expr)
    :_left(m),  _right(expr),
    _left_cache_matrix(NULL),
    rows_(m.rows()), cols_(expr.cols())
  { ParameterCheck(m, expr);
    _right_cache_matrix = new RightCacheMatrixType(expr.rows(), expr.cols());
  }
```

Finally, the fourth constructor creates two caches, each one for one of its operands:

```
  template<class Expr1, class Expr2>
  MultiplicationExpression(const BinaryExpression<Expr1>& expr1, const BinaryExpression<Expr2>& expr2)
    :_left(expr1), _right(expr2),
    rows_(expr1.rows()), cols_(expr2.cols())
  { ParameterCheck(expr1, expr2);
    _left_cache_matrix = new LeftCacheMatrixType(expr1.rows(), expr1.cols());
    _right_cache_matrix = new RightCacheMatrixType(expr2.rows(), expr2.cols());
  }
```

Since the expressions are returned by the operator templates by copy, we need to implement a copy constructor for the multiplication expression. When the expression is copied, the cache variables of the new copy will point to the caches of the old expression. Thus, we need to reset the cache variables in the old expression to NULL, so that its destructor does not destroy the caches:

```
  MultiplicationExpression(MultiplicationExpression& old)
    : _left(old._left), _right(old._right),
    _left_cache_matrix(old._left_cache_matrix),
    _right_cache_matrix(old._right_cache_matrix),
    rows_(old.rows()), cols_(old.cols())
  { old._left_cache_matrix= NULL;
    old._right_cache_matrix= NULL;
  }
```

The destructor deletes the caches, if any:

```
  ~MultiplicationExpression()
  {
    delete _left_cache_matrix;
    delete _right_cache_matrix;
  }
```

Finally, we have the getElement() function which also uses a metafunction to select the most appropriate implementation based on the shape of the operands:

```
    ElementType getElement(const IndexType & i, const IndexType & j) const
    { return MATRIX_MULTIPLY_GET_ELEMENT<LeftType, RightType>::RET::getElement(i, j,
        this, _left, _right, _left_cache_matrix, _right_cache_matrix);
    }

    IndexType rows() const {return rows_;}
    IndexType cols() const {return cols_;}

  private:
    void ParameterCheck(const A& m1, const B& m2)
    { if (m1.cols() != m2.rows())
        throw "argument matrices are incompatible";
    }
};
```

The last class template is BinaryExpression<>. It is derived from its parameter, i.e. ExpressionTemplate. Thus, it inherits Config and the getElement() method from the expression it wraps.

```
template<class ExpressionType>
class BinaryExpression : public ExpressionType
{ public:
    typedef ExpressionType::LeftType LeftType;
    typedef ExpressionType::RightType RightType;
    typedef ExpressionType::Config::MatrixType MatrixType;
    typedef ExpressionType::IndexType IndexType;

    BinaryExpression(const LeftType& __op1, const RightType& __op2)
      : ExpressionType(__op1, __op2)
    {}
```

The following method implements assignment and is called from the assignment operator implementation in Matrix<> (Section 10.3.1.4.4):

```
    template<class Res>
    Matrix<Res>* assign(Matrix<Res>* const result) const
    { MATRIX_ASSIGNMENT<MatrixType>::RET::assign(result, this);
      return result;
    }

    ostream& display(ostream& out) const
    { IndexType r= rows(), c= cols();
      for( IndexType i = 0; i < r; ++i )
      { for( IndexType j = 0; j < c; ++j )
          out << getElement( i, j ) << "   ";
        out << endl;
      }
      return out;
    }
};

template <class Expr>
ostream& operator<<(ostream& out, const BinaryExpression<Expr>& expr)
{ return expr.display(out);
}
```

10.3.1.7.3 Matrix Cache

As stated, matrix multiplication uses a cache to avoid recomputing the elements of an operand expression. We implement the cache as a matrix whose elements are cache elements rather than numbers. A cache element has a variable for storing the cached element value and a flag indicating if the value is in cache or not (in MatrixCache.h):

Module: MatrixCache.h

```
template<class ElementType>
struct CacheElementType
{ bool valid;   //if true, the value is already cached (cache-hit); if false, it isn' t
  ElementType element;
```

```
  CacheElementType() : element(ElementType(0)), valid(false) {}

  CacheElementType(const ElementType& elem)
    : element(elem), valid(false) {}

  bool operator==(const CacheElementType& scnd) const
  {return (valid == scnd.valid && element == scnd.element);}

  bool operator!=(const CacheElementType& scnd) const
  {return (valid != scnd.valid || element != scnd.element);}

  ostream& display(ostream& out) const
  { out << "(" << element << "; " << valid << ")";
    return out;
  }
};

template <class A>
ostream& operator<<(ostream& out, const CacheElementType<A>& elem)
{ return elem.display(out);
}
```

Next, we implement a metafunction which takes a matrix type and returns the corresponding matrix cache type. The only difference between these two types is the element type: the element type of the cache is CacheElementType<> parameterized with the element type of the original matrix type. The cache type derivation involves reading out the description of the matrix type, i.e. DSLFeatures, and deriving a new type from this description, which overrides the inherited element type with the new cache element type, and finally passing DSLFeatures to the matrix generator. In order to be able to do the derivation, we need a little workaround since it is not possible to derive a class from a typename defined by a typedef:

```
struct DerivedDSLFeatures : public DSLFeatures
{};
```

And here is the metafunction:

```
template<class MatrixType>
struct CACHE_MATRIX_TYPE
{ private:
    typedef MatrixType::Config Config;
    typedef Config::DSLFeatures DSLFeatures;

  public:
    //override ElementType:
    struct CachedMatrixDSL : public DerivedDSLFeatures<DSLFeatures>
    { typedef CacheElementType<DSLFeatures::ElementType> ElementType;
    };

    typedef MATRIX_GENERATOR<CachedMatrixDSL, assemble_components>::RET RET;
};
```

10.3.1.7.4 *Implementation of* ***getElement()***

The implementation of getElement() for the addition expression and for the multiplication expression depends on the shape of the operands. Therefore, we will use the same technique as in the case of nonZeroRegion() of ArrFormat in Section 10.3.1.4.2: we implement the method variants as static methods of separate structs and select the structs using metafunctions.

In the case of addition, we provide three algorithms: one general for adding rectangular matrices (which also works in all other cases), one for adding lower triangular matrices, and one for adding upper triangular matrices (in GetElement.h):

Module: GetElement.h

```
struct RectAddGetElement
{ template<class IndexType, class ResultType, class LeftType, class RightType>
```

```
  static ResultType::ElementType
  getElement(const IndexType& i, const IndexType& j,
    const ResultType* res, const LeftType& left, const RightType& right)
  {
    return left.getElement(i, j) + right.getElement(i, j);
  }
};

struct LowerTriangAddGetElement
{
  template<class IndexType, class ResultType, class LeftType, class RightType>
  static ResultType::ElementType
  getElement(const IndexType& i, const IndexType& j,
    const ResultType* res, const LeftType& left, const RightType& right)
  {
    return i >= j ? left.getElement(i, j) + right.getElement(i, j)
            : ResultType::ElementType(0);
  }
};

struct UpperTriangAddGetElement
{
  template<class IndexType, class ResultType, class LeftType, class RightType>
  static ResultType::ElementType
  getElement(const IndexType& i, const IndexType& j,
    const ResultType* res, const LeftType& left, const RightType& right)
  {
    return i <= j ? left.getElement(i, j) + right.getElement(i, j)
            : ResultType::ElementType(0);
  }
};
```

The following metafunction selects the appropriate algorithm: it takes LowerTriangAddGetElement for two lower triangular matrices, UpperTriangAddGetElement for two upper triangular matrices, and RectAddGetElement for all the other combinations:

```
template<class Matrix1, class Matrix2>
struct MATRIX_ADD_GET_ELEMENT
{
  typedef Matrix1::Config::DSLFeatures::Shape Shape1;
  typedef Matrix2::Config::DSLFeatures::Shape Shape2;

  typedef IF< EQUAL<Shape1::id, Shape1::lower_triang_id>::RET &&
        EQUAL<Shape2::id, Shape2::lower_triang_id>::RET,
          LowerTriangAddGetElement,

      IF<  EQUAL<Shape1::id, Shape1::upper_triang_id>::RET &&
        EQUAL<Shape2::id, Shape2::upper_triang_id>::RET,
          UpperTriangAddGetElement,

      RectAddGetElement>::RET>::RET RET;
};
```

The following is the getElement() for the multiplication of two rectangular matrices:

```
struct RectMultiplyGetElement
{
  template<class _IndexType,
    class ResultType, class LeftType, class RightType,
    class LeftCacheType, class RightCacheType>
  static ResultType::ElementType getElement(const _IndexType& i, const _IndexType& j,
    const ResultType* res, const LeftType& left, const RightType& right,
    LeftCacheType* left_cache= NULL, RightCacheType* right_cache= NULL)
  {
    typedef ResultType::Config Config;
    typedef Config::ElementType ElementType;
    typedef Config::IndexType IndexType;

    ElementType result= ElementType(0);
```

```
    for(IndexType k= left.cols(); k--;)
      result+= getCachedElement(i, k, left, left_cache) * getCachedElement(k, j, right, right_cache);
    return result;
  }

  private:
    template<class IndexType, class MatrixType, class CacheType>
    static MatrixType::ElementType
    getCachedElement(const IndexType& i, const IndexType& j,
      const MatrixType& matrix, CacheType* cache)
    {
      if (cache == NULL) return matrix.getElement(i, j);
      else
      {
        CacheType::ElementType tmpCacheElem= cache->getElement(i, j);
        if (!tmpCacheElem.valid)
        {
          tmpCacheElem.element= matrix.getElement(i, j);
          tmpCacheElem.valid= true;
          cache->setElement(i, j, tmpCacheElem);
        }

        return tmpCacheElem.element;
      }
    }
};
```

The variant of getElement() for multiplying two lower triangular matrices and the other variant for two upper triangular matrices are analogous and not shown here.

Here is the metafunction for selecting the algorithms:

```
/****************************** Selecting algorithms for multiplication ******************************/
template<class Matrix1, class Matrix2>
struct MATRIX_MULTIPLY_GET_ELEMENT
{
  typedef Matrix1::Config::DSLFeatures::Shape Shape1;
  typedef Matrix2::Config::DSLFeatures::Shape Shape2;

  typedef IF< EQUAL<Shape1::id, Shape1::lower_triang_id>::RET &&
        EQUAL<Shape2::id, Shape2::lower_triang_id>::RET,
          LowerTriangMultiplyGetElement,

      IF<  EQUAL<Shape1::id, Shape1::upper_triang_id>::RET &&
        EQUAL<Shape2::id, Shape2::upper_triang_id>::RET,
          UpperTriangMultiplyGetElement,

      RectMultiplyGetElement>::RET>::RET RET;
};
```

10.3.1.7.5 Metafunctions for Computing Result Types of Expressions

As you remember, the addition expression and multiplication expression class templates call the metafunctions for computing the matrix result type of addition and multiplication:

```
ADD_RESULT_TYPE<class class MatrixType1, class MatrixType2>
MULTIPLY_RESULT_TYPE<class MatrixType1, class MatrixType2>
```

The metafunctions work as follows: They take the DSLFeatures from each of the operands, compute the result DSLFeatures and call the matrix generator with the result DSLFeatures to generate the result matrix type.

We first need to specify how to compute the result DSLFeatures from the argument DSLFeatures. As you remember, DSLFeatures contains the following features:

ElementType
Shape
Format
ArrayOrder
OptFlag
BoundsChecking
IndexType

Thus, we need to compute these features for the result matrix from the features of the argument matrices. The following features are computed in the same way for both addition and multiplication:

ElementType
Format
ArrayOrder
OptFlag
BoundsChecking
IndexType

ElementType and IndexType are computed using the numeric promote metafunction shown in Figure 178. The table for computing Format is given later. The other features are computed as follows: If the value of the given feature in one argument matrix is equal to the value of the same feature in the other argument matrix, the resulting feature value is equal to the other two values. If this is not the case, the resulting feature value is unspecified_DSL_feature (see Table 110). This is useful, since we can let the matrix generator assign the default value for the unspecified features.

Feature1	Feature2	Result
(value)	(value)	=(value)
*	*	unspecified_DSL_feature

Table 110 *General formula for computing result values of nonmathematical DSL features*

The resulting shape is computed differently for addition and for multiplication. This is shown in Table 111 and Table 112.

Shape1	Shape2	Shape Result
symm	symm	symm
lowerTriang	lowerTriang	lowerTriang
upperTriang	upperTriang	upperTriang
*	*	rect

Table 111 *Computing the resulting shape for matrix addition*

Shape1	Shape2	Shape Result
lowerTriang	lowerTriang	lowerTriang
upperTriang	upperTriang	upperTriang
*	*	rect

Table 112 *Computing the resulting shape for matrix multiplication*

Format depends not only on the Format of the operands, but also on the shape of the result. In particular, adding a lower-triangular matrix to an upper-triangular one yields a rectangular matrix. If the format of both operands is vector, we cannot simply assume vector for the result since a rectangular matrix is better stored in an array. This is specified in Table 113.

Shape Result	Format1	Format2	Format Result
rect	vector	vector	unspecified_DSL_feature
*	(value)	(value)	=(value)
*	*	*	unspecified_DSL_feature

Table 113 *Computing the resulting format*

We start with a metafunction implementing the general formula from Table 110 (in ComputeResultType.h):

Module: ComputeResultType.h

```
template<class Feature1, class Feature2>
struct DEFAULT_RESULT
{ typedef
    IF<  EQUAL<Feature1::id, Feature2::id>::RET,
         Feature1,
         unspecified_DSL_feature>::RET RET;
};
```

RESULT_FORMAT<> implements Table 113:

```
template<class Shape, class Format1, class Format2>
struct RESULT_FORMAT
{
  typedef
    IF<  EQUAL<Shape::id, Shape::rect_id>::RET &&
         EQUAL<Format1::id, Format1::vector_id>::RET &&
         EQUAL<Format2::id, Format2::vector_id>::RET,
         unspecified_DSL_feature,

         IF<  EQUAL<Format1::id, Format2::id>::RET,
         Format1,

    unspecified_DSL_feature>::RET>::RET RET;
};
```

The following metafunction computes result shape for matrix addition (Table 111):

```
//compute result shape for addition (
template<class Shape1, class Shape2>
struct ADD_RESULT_SHAPE
{

  typedef
    IF<  EQUAL<Shape1::id, Shape1::symm_id>::RET &&
         EQUAL<Shape2::id, Shape2::symm_id>::RET,
         symm<>,

    IF<  EQUAL<Shape1::id, Shape1::lower_triang_id>::RET &&
         EQUAL<Shape2::id, Shape2::lower_triang_id>::RET,
         lower_triang<>,

    IF<  EQUAL<Shape1::id, Shape1::upper_triang_id>::RET &&
         EQUAL<Shape2::id, Shape2::upper_triang_id>::RET,
         upper_triang<>,

    rect<> >::RET>::RET>::RET RET;

};
```

ADD_RESULT_DSL_FEATURES<> computes result DSLFeatures from two argument DSLFeatures. We refer to the resulting DSLFeatures as ParsedDSL since it may contain some unspecified features (thus, it has the same form as the DSLFeatures returned by the DSLParser).

```
template<class DSLFeatures1, class DSLFeatures2>
class ADD_RESULT_DSL_FEATURES
{
  private:
```

```
    //ElementType (PROMOTE_NUMERIC_TYPE<> is shown in Figure 178)
    typedef PROMOTE_NUMERIC_TYPE<DSLFeatures1::ElementType, DSLFeatures2::ElementType>::RET
        ElementType;

    //IndexType
    typedef PROMOTE_NUMERIC_TYPE<DSLFeatures1::IndexType, DSLFeatures2::IndexType>::RET
        IndexType;

    //Shape
    typedef ADD_RESULT_SHAPE<DSLFeatures1::Shape, DSLFeatures2::Shape>::RET Shape;

    //OptFlag
    typedef DEFAULT_RESULT<DSLFeatures1::OptFlag, DSLFeatures2::OptFlag>::RET OptFlag;

    //BoundsChecking
    typedef DEFAULT_RESULT<DSLFeatures1::BoundsChecking, DSLFeatures2::BoundsChecking>::RET
        BoundsChecking;

    //Format
    typedef RESULT_FORMAT<Shape, DSLFeatures1::Format, DSLFeatures2::Format>::RET Format;

    //ArrOrder
    typedef DEFAULT_RESULT<DSLFeatures1::ArrOrder, DSLFeatures2::ArrOrder>::RET ArrOrder;

  public:
    struct ParsedDSL
    {
      typedef ElementType ElementType;
      typedef Shape Shape;
      typedef Format Format;
      typedef ArrOrder ArrOrder;
      typedef OptFlag OptFlag;
      typedef BoundsChecking BoundsChecking;
      typedef IndexType IndexType;
    };

    typedef ParsedDSL RET;
};
```

ADD_RESULT_TYPE<> returns the result matrix type for addition. It calls the above metafunction in order to compute the resulting DSL features and the matrix generator to generate the matrix type:

```
template<class MatrixType1, class MatrixType2>
struct ADD_RESULT_TYPE
{
  typedef ADD_RESULT_DSL_FEATURES < MatrixType1::Config::DSLFeatures,
                                    MatrixType2::Config::DSLFeatures> BaseClass;
  typedef MATRIX_GENERATOR<BaseClass::ParsedDSL, defaults_and_assemble>::RET RET;
};
```

The code for computing the result type of multiplication is similar:

```
//this function implements Table 112
template<class Shape1, class Shape2>
struct MULTIPLY_RESULT_SHAPE
{
  typedef

    IF<  EQUAL<Shape1::id, Shape1::lower_triang_id>::RET &&
      EQUAL<Shape2::id, Shape2::lower_triang_id>::RET,
      lower_triang<>,

    IF<  EQUAL<Shape1::id, Shape1::upper_triang_id>::RET &&
      EQUAL<Shape2::id, Shape2::upper_triang_id>::RET,
      upper_triang<>,

    rect<> >::RET>::RET RET;
};
```

```
template<class DSLFeatures1, class DSLFeatures2>
struct MULTIPLY_RESULT_DSL_FEATURES
{
  private:
    //ElementType
    typedef PROMOTE_NUMERIC_TYPE<DSLFeatures1::ElementType, DSLFeatures2::ElementType>::RET
      ElementType;

    //IndexType
    typedef PROMOTE_NUMERIC_TYPE<DSLFeatures1::IndexType, DSLFeatures2::IndexType>::RET
      IndexType;

    //Shape
    typedef MULTIPLY_RESULT_SHAPE<DSLFeatures1::Shape, DSLFeatures2::Shape>::RET Shape;

    //OptFlag
    typedef DEFAULT_RESULT<DSLFeatures1::OptFlag, DSLFeatures2::OptFlag>::RET OptFlag;

    //BoundsChecking
    typedef DEFAULT_RESULT<DSLFeatures1::BoundsChecking, DSLFeatures2::BoundsChecking>::RET
      BoundsChecking;

    //Format
    typedef RESULT_FORMAT<Shape, DSLFeatures1::Format, DSLFeatures2::Format>::RET Format;

    //ArrOrder
    typedef DEFAULT_RESULT<DSLFeatures1::ArrOrder, DSLFeatures2::ArrOrder>::RET ArrOrder;

  public:
    struct ParsedDSL
    {
      typedef ElementType ElementType;
      typedef Shape Shape;
      typedef Format Format;
      typedef ArrOrder ArrOrder;
      typedef OptFlag OptFlag;
      typedef BoundsChecking BoundsChecking;
      typedef IndexType IndexType;
    };

    typedef ParsedDSL RET;
};

template<class MatrixType1, class MatrixType2>
struct MULTIPLY_RESULT_TYPE
{
  typedef MULTIPLY_RESULT_DSL_FEATURES <  MatrixType1::Config::DSLFeatures,
                                          MatrixType2::Config::DSLFeatures> BaseClass;
  typedef MATRIX_GENERATOR<BaseClass::ParsedDSL, defaults_and_assemble>::RET RET;
};
```

10.3.1.7.6 Matrix Assignment

The implementation of matrix assignment depends on the shape of the source matrix (or source expression). In general, our assignment implementations perform two steps: initializing the target matrix with zero elements and assigning the nonzero elements from the source matrix (thus, we have to iterate over the nonzero region of the source matrix). We provide an assignment implementation for assigning a rectangular matrix, lower triangular, and upper triangular (the symmetric case is covered by the rectangular case; see Table 114).

Shape (source matrix)	Assignment
lower_triang	LowerTriangAssignment
upper_triang	UpperTriangAssignment
*	RectAssignment

Table 114 *Selecting the assignment algorithm*

Here is the C++ code for the assignment variants (in Assignment.h):

Module: Assignment.h

```
struct RectAssignment
{
  template<class Res, class M>
  static void assign(Res* res, M* m)
  {
    typedef Res::Config::IndexType IndexType;

    for (IndexType i= m->rows(); i--;)
      for (IndexType j= m->cols(); j--;)
        res->setElement(i, j, m->getElement(i, j));
  }
};

struct LowerTriangAssignment
{
  template<class Res, class M>
  static void assign(Res* res, M* m)
  {
    typedef Res::Config::IndexType IndexType;

    for(IndexType i= 0; i< res->rows(); ++i)
      for(IndexType j= 0; j<=i; ++j)
        res->setElement(i, j, m->getElement(i, j));
  }
};

struct UpperTriangAssignment
{
  template<class Res, class M>
  static void assign(Res* res, M* m)
  {
    typedef Res::Config::IndexType IndexType;

    for(IndexType i= 0; i< res->rows(); ++i)
      for(IndexType j= i; j< res->rows(); ++j)
        res->setElement(i, j, m->getElement(i, j));
  }
};
```

The following metafunction implements Table 114:

```
template<class RightMatrixType>
struct MATRIX_ASSIGNMENT
{ typedef RightMatrixType::Config::DSLFeatures::Shape Shape;

  typedef
      IF<EQUAL<Shape::id, Shape::lower_triang_id>::RET,
        LowerTriangAssignment,

      IF<  EQUAL<Shape::id, Shape::upper_triang_id>::RET,
        UpperTriangAssignment,

      RectAssignment>::RET>::RET RET;
};
```

This concludes the implementation of the demo matrix component.

10.3.1.8 Full Implementation of the Matrix Component

The full implementation of the matrix component specified in Section 10.2 comprises 7500 lines of C++ code (6000 lines for the configuration generator and the matrix components and 1500 lines for the operations).[168] The full implementation illustrated a number of important points:

- *Separation between problem and solution space*: The use of a configuration DSL implemented by a separate set of templates hides the internal library architecture (i.e. the ICCL) from the application programmer. It is possible to change the ICCL (e.g. add new components or change the structure of the ICCL) without having to modify the existing client code. All we have to do is to map the existing DSL onto the new ICCL structure, which requires changes in the configuration generator. To a certain extent, we can even extend the configuration DSL without invalidating the existing client code. The kind of possible changes include

  - enlarging the value scope of existing DSL parameters (e.g. adding new matrix shapes)

  - appending new parameters to existing parameter lists (e.g. we could add new parameters at the end of the matrix parameter list and the structure parameter list; furthermore, we could also add new subfeatures to any of the leave nodes of the DSL feature diagram since all DSL features are implemented as templates).

  In fact, during the performance optimization phase in the development of the matrix component, we had to modify the structure of the ICCL by splitting some components and merging others. Since the functionality scope did not change, we did not have to modify the DSL at all. If we used a library design without a configuration DSL (e.g. as in the STL), we would have to modify the existing application using the library since the applications would hardwire ICCL expressions in their code. Thus, the separation into the configuration DSL and the ICCL supports software evolution.

- *More declarative specification*: The client code can request a matrix using a configuration DSL expression which specifies exactly as much detail as the client wishes to specify. Since the configuration DSL tries to derive reasonable feature defaults from the explicitly specified features, the client code does not have to specify the details that do not concern it directly. Consider the following situation as an analogy: When you buy a car, you do not have to specify all the bolts and wires. There is a complex machinery between you and the car factory to figure out how to satisfy your abstract needs. On the other hand, if you need the car for a race, you may actually want to request some specially-customized parts (provided you have the technical knowledge to do so). The same applies to a configuration DSL: You should be able to specify any detail about the ICCL expression you want. Being forced to specify too much detail (this is the case if you do not have a configuration DSL), makes you dependent on the implementation of the library. If the client code, on the other hand, cannot specify all the details provided by the ICCL, it may well happen that the code will run slower compared to what is possible giving its context knowledge. For example, if you know that the shape of your matrix will not exceed lower triangular shape during the execution, you can specify this property at compile time. If the matrix configuration DSL does not let you do this, the generated component will certainly not run the more efficient lower-triangular algorithms on the matrix data.

- *Minimal redundancy within the library*: Parameterizing out differences allows you to avoid situations where two components contain largely the same code except for a few details that are different. Furthermore, there is a good chance that the "little" components implementing these details can be reused as parameters of more than one parameterized component. Thanks to an aggressive parameterization, we were able to cover the functionality of the matrix component specified in Section 10.2 with only 7500 lines of C++ code. The use of a

configuration DSL enables us to hide some of the fragmentation caused by this parameterization from the user.

- *Coverage of a large number of variants*: The matrix configuration DSL covers some 1840 different kinds of matrices. This number does not take element type, index type, and all the number-valued parameters such as number of rows or the static scalar value into account. Given the 7500 lines of code implementing the matrix component, the average number of lines of code per matrix variant is four. If you count the nine different element types and the nine different index types, the number of matrix variants increases to 149 040. The different possible values for extension, diagonal range, scalar value, etc. (they all can be specified statically or dynamically) increase this number even more.

- *Very good performance*: Despite the large number of provided matrix variants, the performance of the generated code is comparable with the performance of manually coded variants. This is achieved by the exclusive use of static binding, which is often combined with inlining. We did not implement any special matrix optimizations such as register blocking or cache blocking. However, the work in [SL98b] demonstrates that by implementing blocking optimizations using template metaprogramming, it is possible to parallel the performance of highly tuned Fortran 90 matrix libraries.

Unfortunately, the presented C++ techniques also have a number of problems:

- *Debugging*: Debugging template metaprograms is hard. There is no such thing as a debugger for the C++ compilation process. Over time, we had to develop a number of tricks and strategies for getting the information we need (e.g. requesting a nonexistent member of a type in order to force the compiler to print out the contents of a typename in an error report; this corresponds to inspecting the value of a variable in a regular program). Unfortunately, they are not always effective. For example, many compilers would not show you the entire type in the error report if the name exceeds a certain number of characters (e.g. 255). The latter situation is very common in template metaprogramming, where template nesting depths may quickly reach very large numbers (see Figure 186).

- *Error reporting*: There is no way for a template metaprogram to output a string during compilation. On the other hand, template metaprograms are used for structure checking (e.g. parsing the configuration DSL) and other compilation tasks, and we need to be able to report problems about the data the metaprogram works on. In Section 10.3.1.5.2, we saw just a partial solution. This solution is unsatisfactory since there is no way to specify the place where the logical error occurred. In order to do this, we would need access to the internal parse tree of the compiler.

- *Readability of the code*: The readability of template metacode is not very high. We were able to improve on this point by providing explicit control structures (see Section 8.6) and using specialized metafunctions such as the table evaluation metafunction in Section 10.3.1.6. Despite all of this, the code remains still quite peculiar and obscure. Template metaprogramming is not a result of a well-thought out metalanguage design, but rather an accident.

- *Compilation times*: Template metaprograms may extend compilation times by orders of magnitude. The compilation time of particular metacode depends on its complexity and the programming style. However, in any case, the conclusion is that template metaprogramming greatly extends compilation times. There are at least two reasons for this situation:

  - template metacode is interpreted rather than compiled,

  - C++ compilers are not tuned for this kind of use (or abuse).

- *Compiler limits*: Since, in template metaprogramming, the computation is done quasi "as a byproduct" of type construction and inference, complex computations quickly lead to very complex types. The complexity and size limits of different compilers are different, but, in general, the limits are quickly reached. Thus, the complexity of the computations is also limited (e.g. certain loops cannot iterate more than some limited number of times).

- *Portability*: Template metaprogramming is based on many advanced C++ language features, which are not (yet) widely supported by many compilers. There are even differences in how some of these features are supported by a given compiler. Thus, currently, template metaprograms have a very limited portability. This situation will hopefully change over the next few years when more and more compiler vendors start to support the newly completed ISO C++ standard.

In general, we conclude that the complexity of template metaprograms is limited by compiler limits, compilation times, and debugging problems.

```
MultiplicationExpression<class LazyBinaryExpression<class AdditionExpression<class MatrixICCL::Matrix<class
MatrixICCL::BoundsChecker<class MatrixICCL::ArrFormat<class MatrixICCL::StatExt<struct
MatrixDSL::int_number<int,7>,struct MatrixDSL::int_number<int,7>>,class MatrixICCL::Rect<class
MatrixICCL::StatExt<struct MatrixDSL::int_number<int,7>,struct MatrixDSL::int_number<int,7>>>,class
MatrixICCL::Dyn2DCContainer<class MATRIX_ASSEMBLE_COMPONENTS<class
MATRIX_DSL_ASSIGN_DEFAULTS<class MATRIX_DSL_PARSER<struct MatrixDSL::matrix<int,struct
MatrixDSL::structure<struct MatrixDSL::rect<struct MatrixDSL::stat_val<struct MatrixDSL::int_number<int,7>>,struct
MatrixDSL::stat_val<struct MatrixDSL::int_number<int,7>>,struct MatrixDSL::unspecified_DSL_feature>,struct
MatrixDSL::dense<struct MatrixDSL::unspecified_DSL_feature>,struct MatrixDSL::dyn<struct
MatrixDSL::unspecified_DSL_feature>>,struct MatrixDSL::speed<struct
MatrixDSL::unspecified_DSL_feature>,struct MatrixDSL::unspecified_DSL_feature,struct
MatrixDSL::unspecified_DSL_feature,struct MatrixDSL::unspecified_DSL_feature,struct
MatrixDSL::unspecified_DSL_feature>>::DSLConfig>::DSLConfig>>>>,class MatrixICCL::Matrix<class
MatrixICCL::BoundsChecker<class MatrixICCL::ArrFormat<class MatrixICCL::StatExt<struct
MatrixDSL::int_number<int,7>,struct MatrixDSL::int_number<int,7>>,class MatrixICCL::Rect<class
MatrixICCL::StatExt<struct MatrixDSL::int_number<int,7>,struct MatrixDSL::int_number<int,7>>>,class
MatrixICCL::Dyn2DCContainer<class MATRIX_ASSEMBLE_COMPONENTS<class
MATRIX_DSL_ASSIGN_DEFAULTS<class MATRIX_DSL_PARSER<struct MatrixDSL::matrix<int,struct
MatrixDSL::structure<struct MatrixDSL::rect<struct MatrixDSL::stat_val<struct MatrixDSL::int_number<int,7>>,struct
MatrixDSL::stat_val<struct MatrixDSL::int_number<int,7>>,struct MatrixDSL::unspecified_DSL_feature>,struct
MatrixDSL::dense<struct MatrixDSL::unspecified_DSL_feature>,struct Ma...
```

Figure 186 *Fraction of the type generated by the C++ compiler for the matrix expression (A+B)*C*

10.3.2 Implementing the Matrix Component in IP

This section gives a short overview of a prototype implementation of the matrix component in the Intentional Programming System (see Section 6.4.3).[169] The prototype covers only a small subset of the functionality specified in Section 10.2. Its scope is comparable to the scope of the demo matrix component (Sections 10.3.1.2 through 10.3.1.7). More precisely, the prototype covers the following matrix parameters: element type, shape (rectangular, diagonal, lower triangular, upper triangular, symmetric, and identity), format (array and vector), and dynamic or static row and column numbers.

The implementation consists of two IP files:

- interface file containing the declaration of the matrix intentions (e.g. matrix type, matrix operations, configuration DSL parameters and values) and

- implementation file containing implementation modules with rendering, editing, and transforming methods.

The implementation file is compiled into an extension DLL. The interface file and the DLL are given to the application programmer who wants to write some matrix code.

Figure 187 shows some matrix application code written using the intentions defined in our prototype matrix library. The code displayed in the editor is rendered by the rendering methods contained in the extension DLL. The DLL also provides editing methods, which, for example, allow us to tab through the elements of the matrix literal used to initialize the matrix variable mFoo (i.e. it behaves like a spreadsheet).

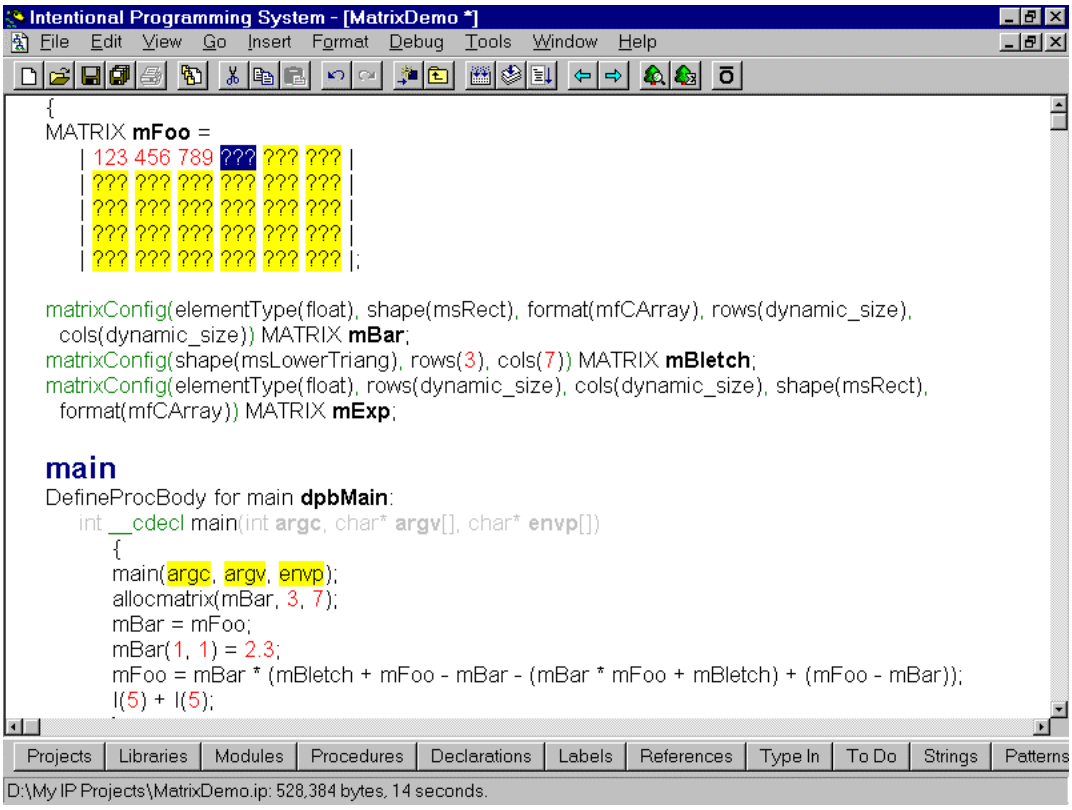

Figure 187 *Sample matrix program using the IP implementation of the matrix component*

In order to appreciate the effect of the displaying methods, Figure 188 demonstrates how the same matrix code looks like when the extension library is not loaded. In this case, the source tree is displayed using the default rendering methods, which are provided by the system. These methods render the source tree in a functional style.

The prototype allows you to declare a matrix as follows:

```
matrixConfig(elementType(float), shape(msRect), format(mfCArray), rows(dynamic_size), cols(dynamic_size))
MATRIX aRectangularMatrix;
```

MATRIX denotes the matrix type. It is annotated by matrixConfig, which specifies matrix parameters. This declaration is quite similar to what we did in the C++ implementation. The main difference is that, in contrast to the C++ template solution, the parameter values are specified by name in any order.

A matrix expression looks exactly like in C++, e.g.:

```
mFoo = mBar * (mBletch + mFoo - mBar - (mBar * mFoo + mBletch) + (mFoo - mBar));
```

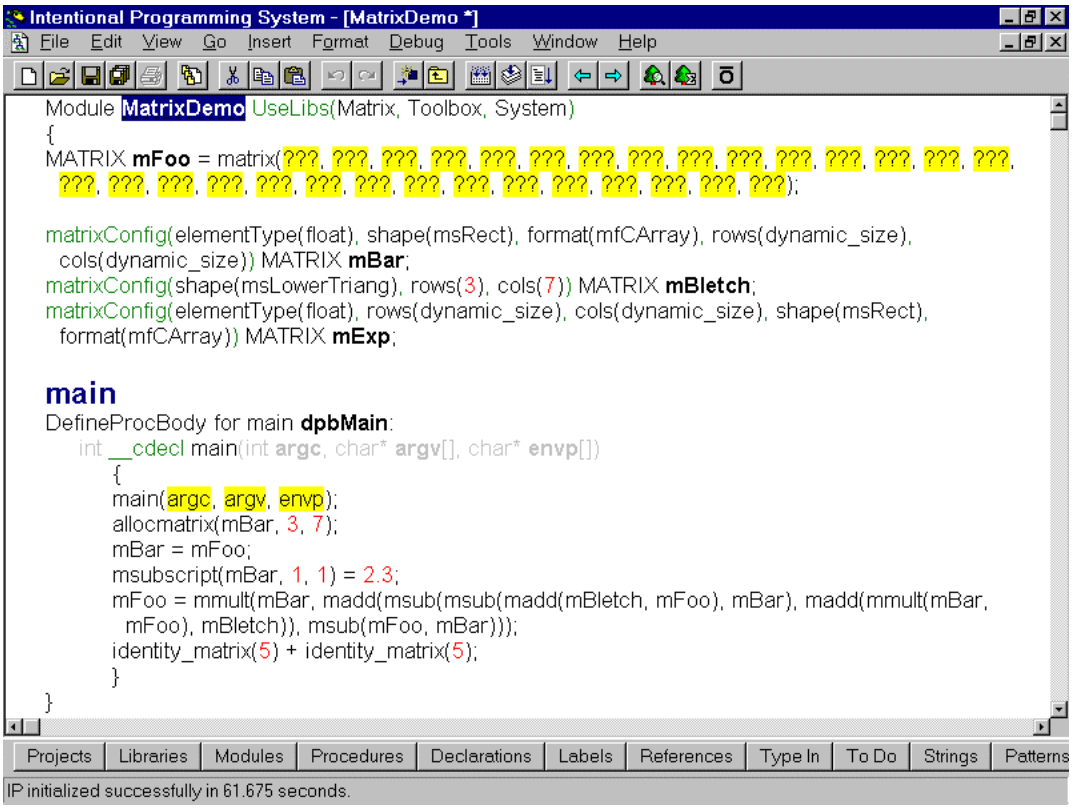

Figure 188 *Sample matrix program (from Figure 187) displayed using the default rendering (mmult is the name of the intention representing matrix multiplication and madd the name of the intention representing matrix addition)*

Table 115 gives an overview of the library implementation.

IP document (i.e. IP file)	Module	Contents
matrix interface (Matrix.ip)	ADTs	declares intentions representing matrix type, matrix literal, and all the matrix operations (multiplication, addition, subtraction, number of rows, number of columns, subscripts, initialization, etc.)
	Config	declares intentions representing the DSL parameters and parameter values (e.g. elementType, shape, format, rect, lowerTriang, etc.)
	VIs	declares virtual intentions for the matrix type (see Section 6.4.3.3)
matrix implementation (MatrixImpl.ip)	RenderingAndEditing	implements the rendering and editing methods for matrix type, matrix literal, matrix operations, and matrix configuration DSL specifications
	MatrixTypeTransforms	implements processing of a matrix configuration (structure check, computing defaults, providing a flat configuration record), generating the data structures for a matrix according (e.g. C structs) to the configuration record
	OperationTransforms	implements checking and transforming matrix expressions, computing the result type of expressions, and generating operation implementations in C
	UI	implements a number of dialog boxes, e.g. for entering matrix configuration descriptions using radio buttons (optional) and for setting the number of rows and columns of a matrix literal

Table 115 *Overview of the IP implementation of the matrix library*

The functionality covered by the module MatrixTypeTransforms corresponds to the configuration generator and the matrix configuration components. However, there are two main differences:

- The generating part in MatrixTypeTransforms uses regular C code (mainly if and case statements) to do the configuration-DSL-to-ICCL mapping.

- Rather than being able to implement the matrix implementation components as class templates (which are convenient to compose), the MatrixTypeTransforms has reduce matrix application code to C, which is much more complex to implement (at the time of developing the prototype, the implementation of C++ in IP was not complete).

The module OperationTransforms implements the functionality covered in the operations section of the C++ implementation (i.e. Section 10.3.1.7). There are functions for computing result types of expressions, for checking the structure and optimizing an expression, and for generating C code.

RenderingAndEditing is the only implementation module that does not have its counterpart in the C++ implementation. This is so since C++ matrix programs are represented by ASCII text and no special support is needed to display them. On the other hand, the IP prototype provides a more natural intention for a matrix literal (i.e. the spreadsheet-like initializer of mFoo) than the textual and passive comma-separated list of numbers provided by the C++ solution (see Section 10.2.5.5). This might seem like a detail, but if we consider adding some specialized mathematical symbols (e.g. Figure 90), the advantages of the IP editing approach outweigh the simplicity of the textual, ASCII-based approach.

Compared to template metaprogramming, implementing the matrix library in IP had the following advantages:

- Metacode is easier to write since you can write it as usual C code.
- Debugging is easier since you can debug metacode using a debugger. You can also debug the generated code at different levels of reduction.
- You can easily issue error reports and warnings. They can be attached to the source tree node that is next to where the error occurred.
- It allows you to design new syntax for domain specific abstractions (e.g. the matrix literal).

There were also two main points on the negative side:

- The transformation scheduling protocol of IP used to be quite limiting (see Section 6.4.3.4) and required complicated and obscure tricks to resolve circular dependencies between intentions. However, as of writing, the scheduling part of IP has been completely reworked to remove these limitations.
- The IP APIs for rendering, editing, and transforming are still quite low level. Also, the fact that we had to reduce to C added a lot of complexity.

A more detailed comparison between the template metaprogramming and the IP approach is given in Table 116. Please note that the comparison applies to the current C++ compilers and the current IP System and may look differently in future. Also, this comparison is done in the context of our programming experiment. There are other features of IP such as code refactoring and code reengineering which are not considered here.

Criterion	Template Metaprogramming	Intentional Programming
complexity limits	The complexity of metaprograms is limited by the limits of current C++ compilers in handling very deeply nested templates.	There are no such limits.
debugging support	There is no debugging support.	Metacode can be debugged using a debugger. Also the generated code can be debugged at different levels of reduction.
error reporting	Inadequate.	Error reports and warnings can be attached directly to source tree nodes close to the locations where the problems occur.
programming effort	Due to the lack of debugging support, error-prone syntax (e.g. lots of angle brackets), and current compiler limits, template metaprograms may require significant programming effort. On the other hand, the code to be generated can be represented as easy-to-configure class templates.	The current IP system also requires significant programming effort. This is due to the low level APIs, e.g. tree editing, displaying, etc. The system provides only few declarative mechanisms (e.g. tree quote and simple pattern matching). Also the unavailability of the C++ mechanisms (classes, templates, etc.) adds programming complexity. On the other hand, the system can be extended with declarative mechanisms at any time. This is more scalable than the template metaprogramming approach.
readability of the metacode	Low.	Due to the (currently) low-level APIs, the readability is also low.
compilation speed	Larger metaprograms (esp. with recursion) may have unacceptable compilation times. Template metaprograms are interpreted.	Since IP has been designed for supporting metaprogramming, there are no such problems as with template metaprogramming (e.g. IP metaprograms are compiled). Nevertheless, compiling a C program using a commercial C compiler is much faster than compiling it using the current version of IP. This situation is expected to improve in future versions of IP.
portability/ availability	Potentially wide available, but better support for the C++ standard is required. The same applies to portability.	The IP system is not yet commercially available. In future, the portability will depend on the availability of such systems and interoperability standards.
performance of the generated code	Comparable to manually written code. The complexity of optimizations is limited by the complexity limits of template metaprograms.	Comparable to manually written code or better. This is so since very complex optimizations are possible.
displaying and editing	ASCII	Supports two-dimensional displaying, bitmaps, special symbols, graphics, etc.

Table 116 *Comparison between template metaprogramming and IP*

10.4 Appendix: Glossary of Matrix Computation Terms

Banded A banded matrix has its nonzero elements within a 'band' about the diagonal. The bandwidth of a matrix A is defined as the maximum of |i-j| for which a_{ij} is nonzero. The upper

bandwidth is the maximum j-i for which a_{ij} is nonzero and j>i. See diagonal, tridiagonal and triangular matrices as particular cases. [MM]

Condition number The condition number of a matrix A is the quantity $||A||^2 * ||A\text{-}1||^2$. It is a measure of the sensitivity of the solution of Ax=b to perturbations of A or b. If the condition number of A is 'large', A is said to be ill-conditioned. If the condition number is one, A is said to be perfectly conditioned. The Matrix Market provides condition number estimates based on Matlab's condest() function which uses Higham's modification of Hager's one-norm method. [MM]

Defective A defective matrix has at least one defective eigenvalue, i.e. one whose algebraic multiplicity is greater than its geometric multiplicity. A defective matrix cannot be transformed to a diagonal matrix using similarity transformations. [MM]

Definiteness A matrix A is positive definite if $x^T A x > 0$ for all nonzero x. Positive definite matrices have other interesting properties such as being nonsingular, having its largest element on the diagonal, and having all positive diagonal elements. Like diagonal dominance, positive definiteness obviates the need for pivoting in Gaussian elimination. A positive semidefinite matrix has $x^T A x >= 0$ for all nonzero x. Negative definite and negative semidefinite matrices have the inequality signs reversed above. [MM]

Dense A dense matrix or vector contains a relatively large number of nonzero elements.

Diagonal A diagonal matrix has its only nonzero elements on the main diagonal.

Diagonal Dominance A matrix is diagonally dominant if the absolute value of each diagonal element is greater than the sum of the absolute values of the other elements in its row (or column). Pivoting in Gaussian elimination is not necessary for a diagonally dominant matrix. [MM]

Hankel A matrix A is a Hankel matrix if the anti-diagonals are constant, that is, $a_{ij} = f_{i+j}$ for some vector f. [MM]

Hessenberg A Hessenberg matrix is 'almost' triangular, that is, it is (upper or lower) triangular with one additional off-diagonal band (immediately adjacent to the main diagonal). A unsymmetric matrix can always be reduced to Hessenberg form by a finite sequence of similarity transformations. [MM]

Hermitian A Hermitian matrix A is self adjoint, that is AH = A, where AH, the adjoint, is the complex conjugate of the transpose of A. [MM]

Hilbert The Hilbert matrix A has elements $a_{ij} = 1/(i+j-1)$. It is symmetric, positive definite, totally positive, and a Hankel matrix. [MM]

Idempotent A matrix is idempotent if $A^2 = A$. [MM]

Ill conditioned An ill-conditioned matrix is one where the solution to Ax=b is overly sensitive to perturbations in A or b. See condition number. [MM]

Involutary A matrix is involutary if $A^2 = I$. [MM]

Jordan block The Jordan normal form of a matrix is a block diagonal form where the blocks are Jordan blocks. A Jordan block has its nonzeros on the diagonal and the first upper off diagonal. Any matrix may be transformed to Jordan normal form via a similarity transformation. [MM]

M-matrix A matrix is an M-matrix if a_{ij} <= 0 for all i different from j and all the eigenvalues of A have nonnegative real part. Equivalently, a matrix is an M-matrix if a_{ij} <= 0 for all i different from j and all the elements of A-1 are nonnegative. [MM]

Nilpotent A matrix is nilpotent if there is some k such that Ak = 0. [MM]

Normal A matrix is normal if A A^H = A^H A, where AH is the conjugate transpose of A. For real A this is equivalent to A A^T = A^T A. Note that a complex matrix is normal if and only if there is a unitary Q such that Q^H A Q is diagonal. [MM]

Orthogonal A matrix is orthogonal if A^T A = I. The columns of such a matrix form an orthogonal basis. [MM]

Rank The rank of a matrix is the maximum number of independent rows or columns. A matrix of order n is rank deficient if it has rank < n. [MM]

Singular A singular matrix has no inverse. Singular matrices have zero determinants. [MM]

Sparse A sparse matrix or vector contains only a relatively small number of nonzero elements (often less than 1%).

Symmetric/ Skew-symmetric A symmetric matrix has the same elements above the diagonal as below it, that is, a_{ij} = a_{ji}, or A = A^T. A skew-symmetric matrix has a_{ij} = -a_{ji}, or A = -A^T; consequently, its diagonal elements are zero. [MM]

Toeplitz A matrix A is a Toeplitz if its diagonals are constant; that is, aij = f_{j-i} for some vector f. [MM]

Totally Positive/Negative A matrix is totally positive (or negative, or non-negative) if the determinant of every submatrix is positive (or negative, or non-negative). [MM]

Triangular An upper triangular matrix has its only nonzero elements on or above the main diagonal, that is a_{ij}=0 if i>j. Similarly, a lower triangular matrix has its nonzero elements on or below the diagonal, that is a_{ij}=0 if i<j. [MM]

Tridiagonal A tridiagonal matrix has its only nonzero elements on the main diagonal or the off-diagonal immediately to either side of the diagonal. A symmetric matrix can always be reduced to a symmetric tridiagonal form by a finite sequence of similarity transformations.

Unitary A unitary matrix has A^H = A-1. [MM]

10.5 Appendix: Metafunction for Evaluating Dependency Tables

The following metafunction evaluates dependency tables (see Section 10.3.1.6). This implementation does not use partial template specialization.

```
/******************************************************************************
      file:         table.h

      author:       Krzysztof Czarnecki, Johannes Knaupp
      date:         August 25, 1998

      contents:     declaration of meta function EVAL_DEPENDENCY_TABLE which
                    finds the first matching entry in a selection table.
******************************************************************************/

#ifndef TABLE_H
#define TABLE_H
```

```
#pragma warning( disable : 4786 )       // disable warning: identifier shortened
                                        // to 255 chars in debug information

#include "IF.H"


//****************************** helper structs *********************************

struct Nil {};

//endValue is used to mark the end of a list
//anyValue represents a value that matches anything
enum {  endValue = ~(~0u >> 1),         // least signed integer value
        anyValue = endValue + 1         // second least int
    };

//End represents the end of a list
struct End
{
  enum    { value = endValue };
  typedef End Head;
  typedef End Tail;
};

// ResultOfRowEval is used a struct for returning two result values
template< int found_, class ResultList_ >
struct ResultOfRowEval
{
  enum    { found = found_ };
  typedef ResultList_ ResultList;
};

//helper struct for error reporting
struct ERROR__NoMatchingTableRow
{
  typedef ERROR__NoMatchingTableRow ResultType;
};

//***************************** syntax elements *********************************

template< class ThisRow, class FollowingRows = End >
struct ROW
{
  typedef ThisRow           Head;
  typedef FollowingRows   Tail;
};

template< int value_, class FurtherCells = End >
struct CELL
{
  enum    { value = value_ };
  typedef FurtherCells  Tail;
};

template< class ThisResultType, class FurtherResultTypes = End >
struct RET
{
  typedef ThisResultType          ResultType;
  typedef FurtherResultTypes      Tail;
  enum    { value = endValue };
};

//************************* metafunction for evaluating a single row ****************************
```

```
template< class HeadRow, class TestRow >
class EVAL_ROW
{  //replace later by a case statement
     typedef HeadRow::Tail HeadTail;
     typedef TestRow::Tail RowTail;

     enum { headValue = HeadRow::value,
               testValue = TestRow::value,
               isLast    = (HeadTail::value == endValue),
               isMatch   = (testValue == anyValue) || (testValue == headValue)
          };

     typedef IF<   isLast,
                   ResultOfRowEval< true, RowTail >,
                   EVAL_ROW< HeadTail, RowTail >::RET
             >::RET ResultOfFollowingCols;

  public:
     typedef IF<   isMatch,
                   ResultOfFollowingCols,
                   ResultOfRowEval< false, ERROR__NoMatchingTableRow >
             >::RET RET;
};

template<>
class EVAL_ROW< End, End >
{
public:
  typedef Nil RET;
};

//****************** meta function EVAL_DEPENDENCY_TABLE **********************

template< class HeadRow, class TableBody >
class EVAL_DEPENDENCY_TABLE
{
     typedef EVAL_ROW< HeadRow, TableBody::Head >::RET RowResult;
     typedef RowResult::ResultList     ResultList;
     typedef TableBody::Tail           FurtherRows;

     enum { found         = RowResult::found,
               isLastRow  = (FurtherRows::Head::value == endValue)
          };

     //this IF is used in order to map the recursion termination case onto <End, End>.
     typedef IF<   isLastRow, End, HeadRow >::RET HeadRow_;

     typedef IF<   isLastRow,
                   ERROR__NoMatchingTableRow,
                   EVAL_DEPENDENCY_TABLE< HeadRow_, FurtherRows >::RET_List
             >::RET NextTry;

  public:
     //returns the whole result row (i.e. the RET cells of the matching row)
     typedef IF< found, ResultList, NextTry >::RET RET_List;

     //returns the first RET cell of the matching row
     typedef RET_List::ResultType RET;
};

template<>
class EVAL_DEPENDENCY_TABLE< End, End >
{
public:
  typedef Nil RET_List;
```

```
};

#endif   // #ifndef TABLE_H
```

The following file demonstrates the use of EVAL_DEPENDENCY_TABLE:

```
/*****************************************************************************
        File:         table.cpp
        Author:       Krzysztof Czarnecki, Johannes Knaupp
        Date:         August 25, 1998

        Contents:   test of meta function EVAL_DEPENDENCY_TABLE which
                    finds the first matching entry in a selection table.
*****************************************************************************/

#include "iostream.h"
#include "table.h"

//a couple of types for testing
template< int val_ >
struct Num
{
  enum { val = val_ };
};

typedef Num< 1 > One;
typedef Num< 2 > Two;
typedef Num< 3 > Three;
typedef Num< 4 > Four;


void main ()
{
  // test with a single return type
  typedef EVAL_DEPENDENCY_TABLE
        //*************************************************************
        <          CELL<   1,    CELL<   2                          > >

        , ROW< CELL<    1,    CELL<    3,      RET<     One     > > >
        , ROW< CELL<    1,    CELL<    3,      RET<     Two     > > >
        , ROW< CELL<    1,    CELL<    2,      RET<     Three   > > >
        , ROW< CELL<    1,    CELL<    3,      RET<     Four    > > >
        //*************************************************************
        > > > > >::RET Table_1;

  cout << Table_1::val << endl; //prints " 3"


  // test with two return types
  typedef EVAL_DEPENDENCY_TABLE
        //*********************************************************************
        <          CELL<    4,          CELL<   7                                          > >

        , ROW< CELL<     3,          CELL<   7,          RET< Three,   RET<  Four  > > > >
        , ROW< CELL<     4,          CELL<   5,          RET<  Four,   RET< Three > > > >
        , ROW< CELL< anyValue,   CELL<   7,          RET<  One,    RET< Two   > > > >
        , ROW< CELL< anyValue,   CELL< anyValue, RET<  Two,    RET< One   > > > >
        //*********************************************************************
        > > > > >::RET_List ResultRow;

  typedef ResultRow::       ResultType ReturnType_1;
  typedef ResultRow::Tail::ResultType ReturnType_2;

  cout << ReturnType_1::val << '  \t'   << ReturnType_2::val << endl; //prints " 1 2"
```

}

10.6 References

[ABB+94] E. Anderson, Z. Bai, C. Bischof, J. W. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1994, see http://www.netlib.org/lapack/lug/lapack_lug.html

[BLAS97] Basic Linear Algebra Subprograms: A Quick Reference Guide. University of Tennessee, Oak Ridge National Labolatory, and Numerical Algorithms Group Ltd., May 11, 1997, see http://www.netlib.org/blas/blasqr.ps

[BN94] J. : Barton and L. R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Applications*. Addison-Wesley, 1994

[Bre98] U. Breymann. *Designing Components with the C++ STL: A new approach to programming*. Addison Wesley Longman, 1998

[CHL+96] S. Carney, M. Heroux, G. Li, R. Pozo, K. Remington, and K. Wu. A Revised Proposal for a Sparse BLAS Tookit. SPARKER Working Note #3, January 1996, see http://www.cray.com/PUBLIC/APPS/SERVICES/ALGRITHMS/spblastk.ps

[DBMS79] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1979

[DDDH90] J. Dongarra, I. Duff, J. DuCroz, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. In *ACM Transactions on Mathematical Software*, vol. 16, 1990, pp. 1-17, see http://www.netlib.org/blas/blas2-paper.ps

[DDHH88] J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. In *ACM Transactions on Mathematical Software*, vol. 14, no. 1, 1988, pp. 1-32, see http://www.netlib.org/blas/blas3-paper.ps

[DLPRJ94] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A Sparse Matrix Library in C++ for High Performance Architectures. In *Proceedings of the Second Object Oriented Numerics Conference*, 1994, pp. 214-218, see ftp://gams.nist.gov/pub/pozo/papers/sparse.ps.Z

[DLPR96] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. IML++ v. 1.2 Iterative Method Library Reference Guide, 1996, see ftp://gams.nist.gov/pub/pozo/docs/iml.ps.gz

[DPW93] J. Dongarra, R. Pozo, and D. Walker. LAPACK++: A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra. In *Proceedings of Supercomputing '93*, IEEE Computer Society Press, 1993, pp. 162-171, see http://math.nist.gov/lapack++/

[Ede91] A. Edelman. The first annual large dense linear system survey. In *SIGNUM Newsletter*, no. 26, October 1991, pp. 6-12, see ftp://theory.lcs.mit.edu/pub/people/edelman/parallel/survey1991.ps

[Ede93] A. Edelman. Large Dense Numerical Linear Algebra in 1993: The Parallel Computing Influence by A. Edelman. In *Journal of Supercomputing Applications*, vol. 7, 1993, pp. 113-128, see ftp://theory.lcs.mit.edu/pub/people/edelman/parallel/1993.ps

[Ede94] A. Edelman. Large Numerical Linear Algebra in 1994: The Continuing Influence of Parallel Computing. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 781-787, see ftp://theory.lcs.mit.edu/pub/people/ edelman/parallel/large94.ps

[FS97] F. Gomes and D. Sorensen. C++: A c++ implementation of ARPACK eigenvalue package. User's Manual, draft version, August 7, 1997, see http://www.caam.rice.edu/software/ARPACK/arpackpp.ps.gz

[GL96] G. Golub and C. van Loan. *Matrix Computations*. Third edition. The John Hopkins University Press, Baltimore and London, 1996

[Hig96] N. Higham. Recent Developments in Dense Numerical Linear Algebra. Numerical Analysis Report No. 288, Manchester Centre for Computational Mathematics, University of Manchester, August 1996; to appear in *The State of the Art in Numerical Analysis*, I. Duff and G. Watson, (Eds.), Oxford University Press, 1997, see ftp://ftp.ma.man.ac.uk/pub/narep/narep288.ps.gz

[ILG+97] J. Irwin, J.-M. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-Oriented Programming of Sparse Matrix Code. XEROX PARC Technical Report SPL97-007 P9710045, February 1997, see http://www.parc.xerox.com/aop

[JK93] A. Jennings and J. McKeown. *Matrix Computation*. Second edition, John Wiley & Sons Ltd., 1993

[KL98] K. Kreft and A. Langer. Allocator Types. In *C++ Report*, vol. 10, no. 6, June 1998, pp. 54-61

[LHKK79] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. In *ACM Transactions on Mathematical Software*, vol. 5, 1979, pp. 308-325

[LSY98] R. Lehoucq, D. Sorensen, and C. Yang. *ARPACK Users' Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. Society for Industrial and Applied Mathematics (SIAM), 1998, see http://www.caam.rice.edu/software/ARPACK/usergd.html

[MM] Matrix Market at http://math.nist.gov/MatrixMarket/, an on-line repository of matrices and matrix generation tools, Mathematical and Computational Science Division within the Information Technology Laboratory of the National Institute of Standards and Technology (NIST)

[MS96] D. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, Massachusetts, 1996

[Neu98] T. Neubert. Anwendung von generativen Programmiertechniken am Beispiel der Matrixalgebra. Diplomarbeit, Technische Universität Chemnitz, 1998

[OONP] The Object-Oriented Numerics Page (maintained by T. Veldhuizen) at http://monet.uwaterloo.ca/oon/

[Poz96] R. Pozo. Template Numerical Toolkit for Linear Algebra: high performance programming with C++ and the Standard Template Library. Presented at the Workshop on Environments and Tools For Parallel Scientific Computing III, held on August 21-23, 1996 at Domaine de Faverges-de-la-Tour near Lyon, France, 1996, see http://math.nist.gov/tnt/

[Pra95] R. Pratap. *Getting Started with Matlab*. Sounders College Publishing, Fort Worth, Texas, 1995

[SBD+76] B. Smith, J. Boyle, J. Dongarra, B. Garbow, Y. Ikebe, V. Klema, and C. Moler. *Matrix Eigensystem Routines – EISPACK Guide*. Second edition, vol. 6 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1976

[Sch89] U. Schendel. *Sparse Matrices: numerical aspects with applications for scientists and engineers*. Ellies Horwood Ltd., Chichester, West Sussex, England, 1989

[SL98a] J. G. Siek and A. Lumsdaine. The Matrix Template Library: A Unifying Framework for Numerical Linear Algebra. In *Proceedings of the ECOOP'98 Workshop on Parallel Object-Oriented Computing (POOSC'98)* , 1998, see http://www.lsc.nd.edu/

[SL98b] J. G. Siek and A. Lumsdaine. A Rational Approach to Portable High Performance: The Basic Linear Algebra Instruction Set (BLAIS) and the Fixed Algorithm Size Template (FAST) Library. In *Proceedings of the ECOOP'98 Workshop on Parallel Object-Oriented Computing (POOSC'98)* , 1998, see http://www.lsc.nd.edu/

[Vel95] T. Veldhuizen. Expression Templates. In *C++ Report*, vol. 7 no. 5, June 1995, pp. 26-31, see http://monet.uwaterloo.ca/blitz/

[Vel97] T. Veldhuizen. Scientific Computing: C++ versus Fortran. In *Dr. Dobb' s Journal*November 1997, pp. 34-41, see http://monet.uwaterloo.ca/blitz/

[Wil61] J. Wilkinson. Error Analysis of Direct Methods of Matrix Inversion. In *Journal of the ACM*, vol. 8, 1961, pp. 281-330