FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Refactoring and Extension of ExaStencils Capabilities for Matrix Datatypes and Operations

Fabian Böhm

Bachelor's thesis

Refactoring and Extension of ExaStencils Capabilities for Matrix Datatypes and Operations

Fabian Böhm

Bachelor's thesis

Aufgabensteller:	Prof. Dr. H. Köstler
Betreuer:	Dr.-Ing. S. Kuckuk
Bearbeitungszeitraum:	1.6.2020 – 1.11.2020

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelor's thesis einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 1. November 2020 . . . .

Abstract

The ExaStencils code generator produces iterative solvers for Partial differential equation (PDE) problems as code written in the C++ programming language. Input for the generator is the problem, formulated in the domain-specific language (DSL) ExaSlang. There, the equations, grids on which the equations are defined, boundary conditions, and the solution algorithm are specified. The generator emits code that executes the algorithm, adapts it to the present hardware and furthermore applies optimizations to it.
In ExaSlang, an interface for matrix data types and operations such as inversion of a matrix is implemented as a prototype already. An implementation for processing matrices in the generator exists as well, which is used by developers of the generator for a wider application scope than only matrices occurring in the input problem. One of them is applying an iterative solver by setting up small local linear systems of equations and solving them under use of matrix datatypes, effectively obtaining solutions for multiple unknowns of the problem at once.
But the present implementation has different problems: it is at parts rudimentary and unstructured. The scope of operations available in the DSL is restricted. During inversion of matrices at generation time, large numbers of nodes in the abstract syntax tree (AST) are produced. This limits the size of matrices that can be inverted and slows down the generation process. Also, the mechanisms to solve small linear systems of equations are not tailored for often occurring types of problems and therefore produce large overheads.
The aim of this work is refactoring and extension of ExaStencils capabilities for matrix data types and operations.
The processing of built-in matrix functions in the generator is reworked to simplify and structure them. Available operations for matrices in ExaSlang are extended by slicing. Optimized matrix inversion algorithms based on the shape of the matrix are implemented, as well as the automatic classification of matrix shapes. Furthermore, the direct solution of small linear systems of equations without calculating an inverse is set up.
The new slicing functions could be applied to rewrite an existing test case maintaining the correct results and simplifying the test case. Shapes of local matrices in real-world applications could be classified successfully. Reduced node counts in the AST, reduced time to generate applications, and reduced run-time of the generated application are achieved for the optimizations in inversion and by direct solution of linear systems. This leads to the capability to invert matrices and solve systems up to a larger size in the generator and faster at run-time of the generated application.

Contents

1 Introduction

Partial Differential Equations (PDEs) are of great importance for engineers and scientists as they describe a wide scope of natural phenomena. PDEs are used to gather information about e.g. heat exchange, mechanical stress, and flow problems. This involves discretizing the PDE with one of various discretization techniques and constructing a linear system of equations. To solve the system and obtain a solution, an iterative solver is employed. The solver calculates an approximate solution for the PDE. The linear system of equations to be solved can easily contain multiple thousands of unknowns, demanding high computational effort to solve. Due to that property, the solution program should optimally exploit problem characteristics and the hardware it runs on. One approach to achieve these two goals are domain-specific languages (DSLs) paired with code generators. Leaving out internal DSLs and concentrating on external DSLs, a DSL can be considered as a distinctive programming language, designed to describe a distinctive domain of problems. It contains language constructs that represent domain features and logic, which makes for an easy, familiar, and concise formulation of problem of the domain. A code generator specialized on the DSL reads the problem formulation and recognizes the present hardware. It transforms the input program to a different representation, an abstract syntax tree (AST), and makes modifications to it. These modifications implement optimizations for the solver to be generated. They optimize in two regards: characteristics of the problem are exploited and the solver is adapted for the present hardware, using all its potential computational capabilities. Knowledge from both hardware and domain experts is used in these optimizations. At last, the generator produces a target program from the problem formulation, which incorporates the optimizations and can be compiled and used [16].
ExaStencils is a research project of multiple universities and institutions, among them computer science chair 10 for system simulation of Friedrich-Alexander-University, in whose course a code generator framework was implemented. The generator reads programs written in the DSL "ExaSlang" and generates optimized C++ code. PDE Problems like e.g. the Navier-Stokes-equations can be formulated in different configurations concerning dimensions, domains, and boundary conditions. The code generator produces an optimized, iterative (multigrid) solver under use of parallelization techniques like OpenMp and MPI. Additionally, the iterative solver is adapted to characteristics of the problem and the present hardware.
ExaSlang also contains language constructs to explicitly use matrices. They exist as a datatype, can be declared as variables, and operations like addition, multiplication or inversion can be executed by built-in functions. Furthermore, fields of matrices can be set up. Further applications that involve the use of matrices are to be realized, for example, PDEs which are formulated vector-wise, such as the optical flow equation [17]. Next to matrices being a language feature of the DSL, they are used within the generation process as well by developers of the generator. For example: To solve for multiple unknowns of the solution linear system at once, a local linear system is set up by constructing and solving a small matrix system of multiple unknowns and components of the PDE's right-hand-side. This is necessary for some applications if point-wise solution algorithms are not applicable. In other cases, it is used as an optimization. Both interfaces of matrices are implemented as a prototype [11][17].
This work aims at refactoring and extension of the code generator's capabilities with regards to matrices.
The generator's mechanisms, which process matrices from ExaSlang to C++, are refactored. This is done by setting up dedicated AST-nodes and abstracting AST-modifications via traits, described in section Dedicated Nodes. Then, the scope of operations available for matrices in ExaSlang is extended. Slicing of matrices offers a starting point for this as it is a widely applicable operation. The implementation of slicing is presented in section Slicing and Bracket Access. The most important bullet point of this work is the extension and optimization of solution operations for small linear systems. Matrix-shape-exploiting algorithms to invert matrices are implemented, targeting Schur,- and block-diagonal shapes. The direct solution of linear systems is set up. These topics are subject of the sections Specialized Inversions and Solving Local Linear Systems, respectively. Automatic classification of such matrix shapes is described in section Matrix Shape Classification. Afterward, the Results achieved by the new implementations are listed, including demonstrations of some extensions to the generator. Prior to an outlook of future work, some concluding remarks are formulated.

2 Background

The background section of this work has to provide an overview of topics of multiple different fields because the implementation's theoretical basis is multidisciplinary. The sections Specialized Inversions and Solving Local Linear Systems include algorithms and definitions from linear algebra. Sections Slicing and Bracket Access and Matrix Shape Classification require background knowledge about parsing. All sections are embedded into the ExaStencils generation process, so its workings are presented.

2.1 Linear Algebra

Some definitions and algorithms from linear algebra form the theoretical basis of sections Specialized Inversions and Solving Local Linear Systems so they will be revised here.

2.1.1 Definitions

Inverse matrix

For a regular matrix $A \in \mathbb{R}^{n\times n}$, the corresponding *inverse* matrix $A^{-1} \in \mathbb{R}^{n\times n}$ is the one for which $A * A^{-1} = I$ holds, where I is the unit matrix of size $n \times n$ and $*$ the matrix multiplication [18, p. 51].

Blockdiagonal matrix

A *blockdiagonal* matrix is a quadratic matrix, containing quadratic matrix blocks on the main diagonal. The remaining entres are zero. It is of the form:

$$\begin{pmatrix} A_1 & 0 & \cdots & 0 \\ 0 & A_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \\ 0 & \cdots & 0 & A_n \end{pmatrix}$$

The inverse of a blockdiagonal matrix is again a blockdiagonal matrix, put together of the inverted main diagonal blocks:

$$\begin{pmatrix} A_1 & 0 & \cdots & 0 \\ 0 & A_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \\ 0 & \cdots & 0 & A_n \end{pmatrix}^{-1} = \begin{pmatrix} A_1^{-1} & 0 & \cdots & 0 \\ 0 & A_2^{-1} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \\ 0 & \cdots & 0 & A_n^{-1} \end{pmatrix}$$

Schur complement

For a matrix $M \in \mathbb{R}^{(n+m)\times(n+m)}$ and matrix blocks $A \in \mathbb{R}^{n\times n}$ and $D \in \mathbb{R}^{m\times m}$ in M, the following holds:

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} + A^{-1}BS^{-1}CA^{-1} & -A^{-1}BS^{-1} \\ -S^{-1}CA^{-1} & S^{-1} \end{pmatrix}$$

S is called the *schur-complement* of A and is defined as:

$$S = D - CA^{-1}B \tag{2.1}$$

It arises when bringing the block matrix M to upper triangle form by multiplying it with a lower triangular matrix:

$$\begin{pmatrix} I_n & 0 \\ -CA^{-1} & I_m \end{pmatrix} * \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} A & B \\ 0 & D - CA^{-1}B \end{pmatrix}$$

I_m, I_n denote unit matrices of size m and n, respectively [19, p. 18].
If applicable, that is, if the inverse of A and the inverse of S exist, the schur-complement transforms the problem of inverting a matrix of size $(n+m) \times (n+m)$ to inverting a $m \times m$ matrix and a $n \times n$ matrix. If A is of a certain form, e.g. a blockdiagonal matrix, the inversion is reduced in complexity. This is often used for the applications described in section Specialized Inversions and section Fields and `solve locally`.

Cofactor matrix

The inverse of a $n \times n$ matrix A can also be calculated by multiplying its inverse determinant with the transposed *matrix of cofactors* C:

$$A^{-1} = \frac{1}{|A|} C^T = \frac{1}{|A|} \begin{pmatrix} C_{11} & C_{21} & \cdots & C_{n1} \\ C_{12} & C_{11} & \cdots & C_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ C_{1n} & C_{2n} & \cdots & C_{nn} \end{pmatrix} \tag{2.2}$$

Every C_{ij} is calculated by: $(-1)^{i+j} M_{i,j}$, with $M_{i,j}$ being the determinant of the matrix, arising when deleting the ith row and jth column of A. The complexity of the method is $n!$, making it seldom usable for larger matrices, but useful for very small matrices [2].
This definition is used to calculate small inverse matrices at compile-time in the state of the generator previous to this work. It is evaluated in section Results together with methods implemented in this work.

2.1.2 LU decomposition

The *LU-decomposition* is a factorization method to decompose a matrix A to a product of matrices: $A = LU$, where L is a lower triangular matrix and U is an upper triangular matrix. For $2x2$-matrices this means:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} * \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix}$$

To make the decomposition unique, usually the lower triangular matrix L is defined as a lower unit triangular matrix, where the main diagonal entries are set to one. LU-decomposition can be seen as the matrix view of gaussian elimination. To zero out one column of A, it is multiplied by a *gaussian transformation matrix* M. To obtain a upper triangular matrix U, A can be multiplied with gaussian transformation matrices for all n columns: $M_{n-1}...M_2 M_1 A = U$. Then L can be viewed as a combination of gaussian transformation matrices [10, p. 115]:

$$A = LU \tag{2.3}$$

$$L = M_1^{-1}...M_{n-1}^{-1} \tag{2.4}$$

Suppose $M \in \mathbb{R}^{n \times n}$, then the factorization is computed row wise, within the input matrix A by algorithm 2.1.

Algorithm 2.1 LU decomposition

```
1: function LUDECOMP(A)
2:     for $i \leftarrow 0, n$ do
3:         for $j \leftarrow i+1, n$ do
4:             $A(j,i) \leftarrow A(j,i)/A(i,i)$
5:             for $k \leftarrow i+1, n$ do
6:                 $A(j,k) \leftarrow A(j,k) - A(j,i) * A(i,k)$
7:             end for
8:         end for
9:     end for
10: end function
```

In line 4-6; the row updates (`k`-loop) of gaussian elimination are evident, which are applied on every row below the diagonal (`j`-loop) of every column (`i`-loop). The LU-decomposition of a matrix $M \in \mathbb{R}^{n\times n}$ counts $\frac{2}{3}n^3$ floating point operations (flops) [10, p. 116].
In practice, avoiding numerical instability is achieved by *partial pivoting*, that is a swapping of rows or columns in each step of the outermost loop with the row or column with the largest value at that position. The LU-decomposition is then given as: $PA = LU$, where P is the permutation matrix, holding information about the row permutations. Pseudo code for pivoting by row permutation is given in algorithm 2.2.

Algorithm 2.2 LU decomposition with row permutations

```
 1: function LUPDECOMP(A,P)
 2:     for $i \leftarrow 0, n$ do
 3:         $imax \leftarrow max(|A(i:n, i)|)$
 4:         if $imax != i$ then
 5:             swap $A(imax, :), A(i, :)$
 6:             swap $P(i), P(imax)$
 7:         end if
 8:         // ... rest of LU-decomposition
 9:     end for
10: end function
```

Here, P is saved as just an array, as there are only row exchanges. It saves for each index, which other index should be accesses instead in case the row is swapped. An access to the matrix A is formulated as A(row index, column index), i:n denotes all entries from i to n and : all entries of the row or column. In line 2, the row with the largest absolute value of the ith column is determined and its index is saved. In line 4, a check is implemented: if the row with the highest value in the ith, that is the current column is not the ith row, the two are exchanged. The row permutations are done both in the matrix and in the permutation array, to store the information for later use when solving linear systems by LU-decomposition.
Suppose a linear system has to be solved, given as: $Ax = b$. A denotes the system matrix, x the unknowns of the system and b the right-hand-side vector. Then, A can be LU-decomposed and two triangular systems arise:

$$\begin{aligned} Ax = b &= LUx \\ Ly &= b \\ Ux &= y \end{aligned} \tag{2.5}$$

which can be solved for x by forward and backward substitution.
The pseudo code of these is given in listings 2.3 and 2.4. If the system matrix of the system is pivoted in the LU-decomposition, also the right hand side is permutated. Therefore, the permutation matrix P is used to access it.

Algorithm 2.3 Pseudo code of forward substitution

```
1: function FORWARDSUB(L,y,b,P)
2:     for $i \leftarrow 0, n$ do
3:         $y(i) \leftarrow b(P(i))$
4:         for $j \leftarrow 0, i$ do
5:             $y(i) \leftarrow y(i) - L(i,j) * y(j)$
6:         end for
7:         $y(i) \leftarrow y(i)/L(i,i)$
8:     end for
9: end function
```

Algorithm 2.3 describes solving the ith equation of the triangular system $Ly = b$ for the ith component by reordering, where the `i`-loop iterates from top to bottom. That way, results of `x`-entries below the current `i` are used to solve the `i`th equation for the `i`th component of `x`. Algorithm

2.4 describes solving the ith equation of the triangular system $Ux = y$ for the ith component by reordering, where the `i`-loop iterates from bottom to top.
In contrast to forward substitution, results of `x`-entries above the current `i` are used to solve the

Algorithm 2.4 Pseudo code of backward substitution

```
function BackwardSub(U,x,y,P)
    for $i \leftarrow n-1, 0$ do
        $x(i) \leftarrow y(P(i))$
        for $j \leftarrow 0, i$ do
            $x(i) \leftarrow x(i) - U(i,j) * x(j)$
        end for
        $x(i) \leftarrow x(i)/U(i,i)$
    end for
end function
```

`ith` equation for the `ith` component of `x`. Both forward and backward substitution take n^2 flops to execute [10, p. 106-107].
The LU-decomposition with row permutation is used in this work to directly solve small linear systems, described in section Solving Local Linear Systems.
Furthermore the LU-decomposition can be used to calculate the inverse of a matrix. To do that, a system with a right hand side matrix $I \in \mathbb{R}^{n \times p}$, which is a unit matrix, is solved for an unknowns matrix $X \in \mathbb{R}^{n \times p}$:

$$AX = I = LUX \tag{2.6}$$

Following the definition of an inverse matrix, the solution matrix X is the inverse of A. This way of inverting small matrices is used in section Specialized Inversions to invert matrices.

2.1.3 Iterative Multigrid

Multigrid methods to solve PDE systems apply iterative solvers on meshes of different sizes and aim for optimal convergence using relaxation techniques like Gauss-Seidel. They exploit the capability of these relaxation techniques to smooth highly oscillatory parts of the error and residual of the problem rapidly. The non-oscillatory parts of the error are mapped to higher oscillation on a coarser grid by intergrid-operators and smoothed by applying relaxation there. This procedure can be repeated several times. Afterwards, the solution of the finest grid is corrected by errors calculated on coarser grids. This leads to rapid convergence. A multigrid solver consists of several parts:

- a hierarchy of grids with different mesh sizes
- a relaxation technique or *smoother*, e.g. Jacobi or Gauss-Seidel, to reduce highly-oscillatory parts on each grid
- operators to map between grids of different sizes
- a coarse grid solver to obtain a solution on the coarsest grid of the grid hierarchy

[15, ch. 13]
ExaStencils mostly generates multigrid solvers for given problems. Even though the detailed workings of multigrid solvers are not relevant for this work, sometimes parts of them are mentioned and the language constructs of section Specialized Inversions and Solving Local Linear Systems are applied within the smoother of the generated multigrid solver.

2.2 DSLs and Code Generators

This section gives a short overview of the non-mathematical background for this work. ExaStencils consists of a domain-specific language (ExaSlang) and a *code generator*, which can understand and translate programs written in ExaSlang to C++. A code generator reads an input program formulating a problem, decides how to solve it effectively, and generates a target program that does the solving. A code generator is roughly comparable to a general-purpose language compiler, but is generally less complex, as the input programs are only a specific class of applications. A general-purpose compiler has the aim to process all kinds of applications. The process of code generation is composed of the steps *lexing*, *parsing* and generation of the target program. These steps are described in Code Generator Workflow. Firstly, domain-specific languages are discussed a .

2.2.1 Domain-Specific-Languages

As mentioned in the introductory part, a DSL can be viewed as a small restricted programming language, focused on a particular domain of problems. It should be able to describe all aspects of the problem, and nothing that is not part of the domain. The domain of problems ExaSlang describes is PDE problems with certain restrictions. One concept that PDEs include is equations. These are represented directly as language constructs, e.g. an object for a mathematical equation. Furthermore, also processes of the domain can be part of the language: one part of the domain PDE problems is solving an equation, so there exists a statement in the DSL that commands exactly that. In contrast to general-purpose languages (GPLs), the designer only has one type of application in mind when designing a DSL and most syntactical elements have their predestined, concentrated use area within this application. That means that many of the DSL's keywords are there for a specific reason, and only for this reason. The solve-equation command can not be applied for anything else than solving an equation.
Every programming language has its keywords and delimiters (symbols separating keywords and information the user provides). Together with them, it is strictly defined how a user may combine them to form programs and how he or she may not. These *syntactical rules* are implemented in the language's *parser*, which is part of its compiler. If one tries to write a for-loop using python syntax and compile it with the C++-compiler g++, g++ will directly complain about misplaced colons. Implementing a new parser for a DSL allows naming domain-specific concepts directly with built-in keywords. In ExaSlang, e.g. equations come as a built-in construct and can be used without further effort, similar to built-in datatypes in a general-purpose language. The same way a programmer declares an `int i`, he can declare an `equation eq` [14, ch. 2].
Having access to the compilation process of a program also enables large scale modifications. The abstract action of solving an equation can be formulated in a couple of lines in a DSL like ExaSlang. But the program that has to execute this action contains lots of effort and computation. These are produced within the compilation process of a code generator. To illustrate this: An ExaSlang program may have only 250 lines, but the generated C++ application has 1500 lines.

2.2.2 Code Generator Workflow

This section deals with the generation of the *target program*, defined in the target language from the *source program* defined in the source language, which is separated into three steps: *lexing*, *parsing* and *modifications and additions*. Their interaction is depicted in figure 2.1.

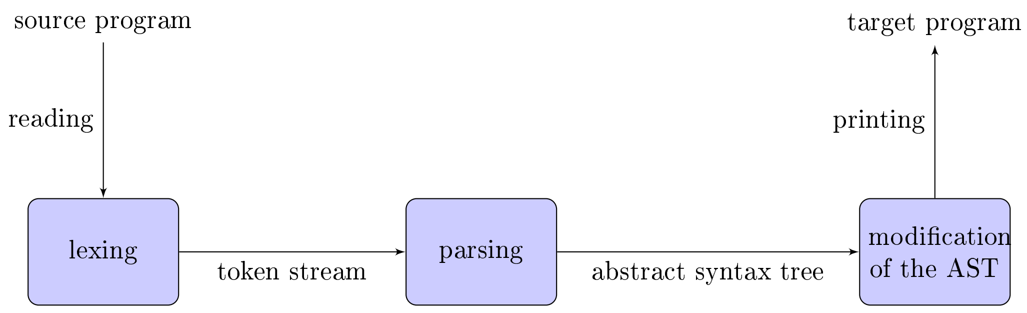


Figure 2.1: Code generator workflow

A source program is read and transformed into a *token stream* by the lexer, which is a series of meaning carrying units. That serves the purpose of making it easier for the parser to perform further processing. The parser brings the input program yet to another form, the *abstract syntax* *tree* (AST). Furthermore, the parser checks for correct language syntax while building the AST. An AST is a tree-like data-structure that enables modifications and additions to the program. These, as well as printing the target program once all modifications and additions are finished and the program is well-formed, are summarized under *modifications and additions* [11, ch. 2]. To understand the parser type used in ExaStencils, also other concepts are needed:

- a *context-free grammar*, that defines the language syntax
- a *statement* is a single instruction expressing some action to be carried out. Programs mostly consist of a series of statements. Examples are assignments and variable declarations, as they completely define one closed action of the program. They connect multiple expressions.
- an *expression* is like a statement a syntactic element of the programming language, that can be evaluated to determine its value. It is a combination of string,- and numerical constant, arithmetic operators, function calls, and possibly many more. Most of the time expressions are used in statements after evaluation, as they do not necessarily define one action of the program by themselves.

These three concepts will be described or used in this section as well.

Lexing

The first step is reading the source program, which the lexer carries out. It separates the input program into a stream of *tokens*. One token is a meaning carrying unit. It consists of a *tokenname* and an additional attribute. The token name specifies one type of element in the source program. For instance, an *identifier* is the name variables and functions have in the source program. The attribute is used to store exact values found in the source program for a token with a certain type defined by its token name. An example token would be <identifier, 'a'> for naming a variable `a`. Besides the lexer's main task, producing a token stream, it also removes commentary and whitespaces from the source program. In the end, the resulting token stream is handed to the *parser*, who processes the token names and attributes further [1, ch. 3].
Background knowledge about lexing was not important for this work, but as it is part of the ExaStencils parsing-process, a short overview was added for completeness.

Syntax Definition

A language's syntax can be seen as the right way to formulate programs in this language. It Defines how keywords, Delimiters, and user-provided data can be combined for a well-formed program in the programming language. They are usually written down as a *context-free grammar*, which is basically a set of rules that say: the user of the language may formulate a command in this or that way. Concretely, the rules are *production rules*. Each production rule determines what form a *non-terminal* can have. Following this, non-terminals are elements of the language that can be produced by a combination of other elements, e.g. an assignment is produced by multiple expressions connected by `=`. *Terminals* are not specified by a production rule but are e.g. single characters like one bracket or keywords like `for`. They are called terminals because they are defined by themselves and not by a combination of other elements in the language. Both non-terminals like variable declarations and terminals like a curly bracket are elements of the language.
In ExaStencils, *combinatory parsers* are used, which are described in section Parsing with Scala Combinators in detail. One of their characteristics is the fact that when declared, they look very similar to the formulation of the grammar they represent. A part of this work was writing such combinatory parsers, which is effectively expanding the language syntax by non-terminals.

Abstract Syntax Tree

The abstract syntax tree is a data structure holding the hierarchical structure of the source program in tree-form. One single node of the AST stores information about one syntactical element of the language. The "abstract" stems from the fact that not every detail of the source code is part of the tree but rather only structural and content-related elements. For example, a ";"-delimiter does not appear in the AST explicitly, but a variable declaration does. The tree consists of nodes and

edges connecting the nodes. A connection by an edge resembles an affiliation relation. E.g. node for a data type of a variable declaration is connected and affiliated to the node for the variable declaration. The tree structure starts with a single node, that has no parent nodes, the *root*. A nodes *parent* is connected to it in the direction of the root. A nodes *child* is connected to it in opposite direction, from the root. A node only has one parent but can have multiple children. If two nodes have the same parent, they both have the same level in the hierarchical structure of the tree and are listed from left to right. Nodes without further children are *leaf nodes*. As a program consists of a collection of statements and scopes, usually the root of an AST is a statement-node or something similar. Leaf nodes are terminals. In figures, child nodes stretch out downwards from the root, which is on top. The parser transforms the given token stream from the lexer according to the rules defined in the languages grammar into an AST.
An example AST is given in 2.2. `decl` is the root node of this small AST, `int` a leaf node and `+`

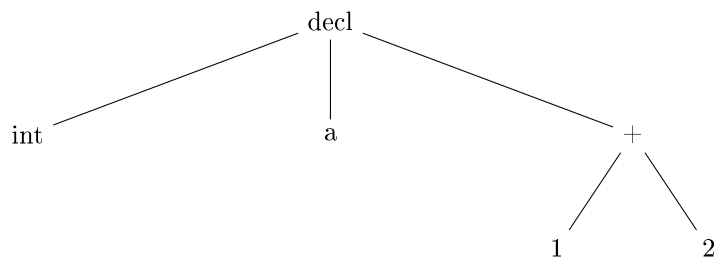


Figure 2.2: AST structure of a variable declaration

an inner node. The hierarchical structure of the program is represented here with the fact that one variable declaration consists of three elements, a data type, a name, and an initialization expression, which are on the same level in the AST,extending from its superpositioned syntactical element, the declaration [1, ch. 2].
In ExaStencils, all AST nodes are instances of classes or traits within one class hierarchy. This is described in detail in section ExaStencils Type Hierarchy. The root node of the AST in figure 2.2 is an instance of a class `variable declaration`, the initialization of an instance of class `addition`. The AST has several useful properties that make the transformation of a program to it worth the effort:

- straight-forward program modification and replacement of nodes
- description of the program without unnecessary elements like delimiters and punctuation
- inspection of the program via tree-traversal to implement semantic and type-related checks

Many of the tasks of this work are implemented as additions and modifications to the AST during its traversal. As the AST is the main platform and data structure the work took place on, it is important to be familiar with them.

Parsing

Parsing is the syntactic analysis of an input token stream with regards to a language's grammar. ExaStencils uses a top-down parser. In a nutshell, this means that the AST is built from root to leaves. Top-down parsers process the input token stream from left to right. For each non-terminal token, they try to determine a fitting production rule. This is done by matching sequences of tokens with the rule's body. Successful matches are added to the AST as nodes. Doing this for all tokens, a parse tree is built, while also checking for correct syntax [1, ch. 4].

ExaStencils parser is a special type of top-down parser. This section gives background to the work described in section Solving Local Linear Systems and Slicing and Bracket Access.

AST Traversal

The AST makes it easier to apply checks, optimizations, and additions to the program before printing out the target program code. In the following, some examples of such actions are presented. Furthermore, the pattern used to traverse an AST is explained. Here, two concepts are essential:

- Looking at one single node, its type, attributes, and connections to child nodes is called *visiting* a node.
- Visiting and subsequently modifying certain nodes is called *transformation*.

Transformation traverse, visit and modify nodes [11, ch. 2]. In the following, some steps taken in the generation process are implemented as transformations are given.
An important check in a code generator or compiler is to find obvious errors like variable accesses to variables which are not defined. Such a check is implemented via transformations *traversing* the AST, which is visiting all nodes and trying to validate all variable accesses by asking: does the variable this access is referring to exist at this point in the program flow?
Optimizations, such as the elimination of common subexpressions in two statements as depicted in listing 2.1 are also implemented by searching for matching cases in the AST while traversing it.

```
f = b * c + d
g = a * b * c
// modifications of the AST are conceptually equal
// to reordering the statements:
tmp = b * c
f = tmp + d
g = a * tmp
```

Listing 2.1: Common subexpression elimination example

In code generators like ExaStencils, often a single statement of the source program is expanded to multiple statements or functions in the target program. E.g. a reduction statement, that is two keywords and a targeted field or array in ExaSlang, will become a declaration for the buffer holding the reduction's result and a loop over the field with additions of field entries to the buffer in its body. These can be implemented by searching for reduction-statement-nodes in the AST and replacing them with new nodes that resemble the targeted functionality. This process is illustrated in section Run-time and Compile-time Execution.
Transformations traverse the AST by *depth-first-search* (DFS). DFS starts at the root node and always visits children of nodes not visited yet. This way, nodes farthest away from the root are visited as quickly as possible. When multiple not-yet-visited child nodes exist, the traversal of these child nodes continues from left to right. Figure 2.3 illustrates this scheme.

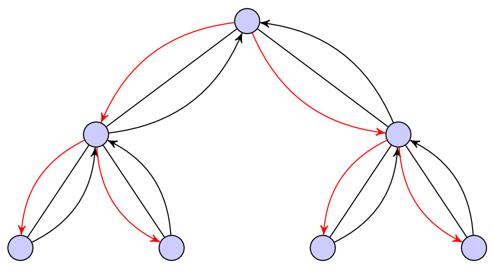


Figure 2.3: Traversal scheme and C-like code of DFS

Red arrows here resemble a recursive call with a child of node `n`, formulated in line 3. Black arrows a return to a higher level call of `dfs` after the current level `dfs` ended in line 7 [1, ch. 2].

Prettyprint

After all validating checks in the AST, optimizations, and additions of nodes to produce target code are done, the AST is transformed back to code. Every node existing up until now knows what form it has to take in the target code, e.g. a `for`-loop node knows that it has to print the `for`-keyword followed by the initialization, condition, incremented and body-statements it holds. As at this point the AST represents a well-formed program, all nodes print their in-code-form to the target files [11, ch. 4].

2.2.3 Run-time and Compile-time Execution

Some kinds of calculations can already be executed within the generation process, at *compile time*. When all operands of an addition are available as constant values within the AST, these operands can be replaced with a constant value node, that is the result of the addition. A constant value node is a node that was added to the AST for a numerical literal in the source program, such as the right-hand side expression of this example code line:

```
a = 2 + 4;
```

Listing 2.2: a short codeline

Also, function calls in the source program can be resolved to the calculation they define, even if the operands are not constant:

```
a = 2;
a_inv = inverse(a);
```

Listing 2.3: another short codeline

Already in the compilation process, the inverse call can be replaced by `1.0/a` [5].
In contrast to compile-time execution, *runtime execution* is the execution of a calculation at runtime of the generated, compiled application. A calculation executed at runtime can arise from two sources: either the user formulated the calculation explicitly in the source program, which is then translated to the target program, compiled, and executed. The other alternative is the generation of a calculation. To do that, nodes which represent the calculation are added to the AST, printed, compiled, and executed. They replace a single call or statement in the source program. For instance, to generate a +-reduction of an array, a variable declaration of the reduction variable and a `for`-loop node must be added to the AST, whose iteration space matches the size of the array. Additionally, a "+="-assignment to the reduction variable with the current array entry is put into the loop body. Figure 2.4 visualizes this: the left AST shows the indication of a reduction statement, which is added to the AST in the parsing process, while the right figure depicts the resulting AST after generation of the reduction-calculation with a collection of statements. `access(id,type,index)` denotes an access at the current point of iteration `index` to the object with identifier `id` and type `type`. The generated statements to calculate a reduction of the int-array `a` replace the reduction node on on the left side.

2.3 ExaStencils

This part outlines the general workings of the ExaStencils code generator. Additionally, it describes the processes and concepts of ExaStencils that served as background and starting point for this work.

2.3.1 From ExaSlang to C++

Layer Overview

The DSL ExaSlang is divided into four hierarchical layers, called layer 1, layer 2, layer 3, and layer 4, with a different focus, syntax, and language constructs. Each layer of the language can be

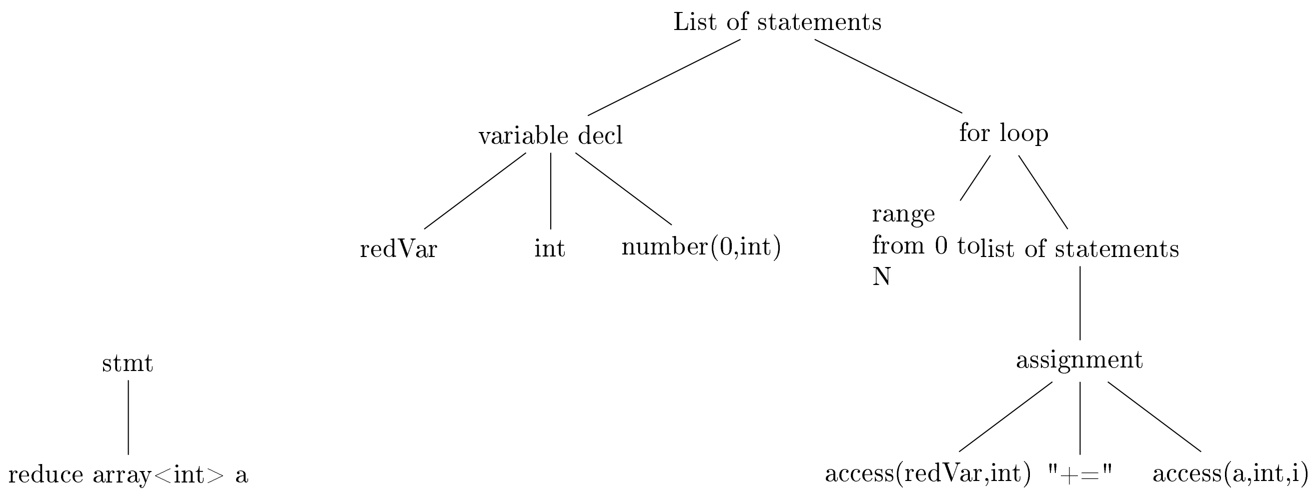


Figure 2.4: generation of code for a reduction

considered its own DSL. ExaSlang also possesses 4 lexers and parsers, one for each layer. LX is used synonymously for L1-4 as an abbreviation for the layers in the following. Each layer conforms to the habits of different professions that users might belong to. The level of abstraction rises with descending layer number. A problem can be formulated in one of the layers and is transformed after lexing and parsing it, to a problem of the layer of the next higher number, and finally from layer 4 to the intermediate representation (IR). The data structure in which the program is present is for all layers a layer-specific AST, with nodes of classes inheriting from layer-specific root classes. Layer 4 describes problems more abstractly than C++-code but is the most detailed and concrete layer of ExaSlang. Complete applications can be described on L4. It contains language constructs to make the formulation of a PDE-problem easy. Important constructs of L4 are fields. L4 fields hold the values of the discrete PDE domain or serve as auxiliary buffers. Besides that, boundary conditions are defined for each field, such as Neumann or Dirichlet. Furthermore, other PDE related constructs are built into L4: stencils, equations and solve statements. L4 is characterized by a computer science focused view. Next to discretized PDE constructs and the solver configuration, data layouts and iteration spaces can be manipulated. For fields, this includes determining the datatype per gridpoint and its localization, such as cell-localized or vertex-localized. Most of the operations to solve a PDE are in a way an iteration over a field, containing value modifications of field entries. Another characteristic of L4 is the possibility of declaring communication points in an MPI configuration. Also, functions are part of L4. They can be declared and called similar to functions in general-purpose programming languages. All other layers map to L4. Matrices are only a part of L4 of ExaSlang and this work modifies only the parsing and generation process of L4-programs [11, ch. 3][12].

Pipeline to generate a C++-Solver

After a program is formulated on one of the layers, a layer-specific lexer and parser produce the AST of that layer. After parsing, many transformations are applied by a *layer handler*. There is a layer handler for each layer. Then, the AST is transformed into the next layer in the pipeline, which is LX + 1. This happens via a single traversal calling an interface inherited *progress*-method for all nodes, transforming every node in a specific way to its counterpart of the next layer. Some nodes are also completely replaced by multiple new nodes. Here L4 is an exception: the L4 AST is progressed to the intermediate representation, which can be considered as another layer between L4 and the generated C++-program, where most of the large scale transformations and optimizations of the program take place. The last step in the pipeline is a prettyprint call for all nodes in the IR AST that produces the target C++ program. This way e.g. an ExaSlang L3 program becomes an L3 AST, then an L4 AST, an IR AST, and at last a compilable C++ program. In the current state of the framework, further files besides the program specification are necessary to provide all information needed for the generation process. Namely a *plattform file*, which defines hardware traits, a *settings*

file for listing input, debug and output files and a *knowledge file* to specify macroscopic parameters in the generation process, like if inversions of matrices should be calculated at run-time or compile-time.
The ExaStencils generation process for an L4 program is outlined in figure 2.5.

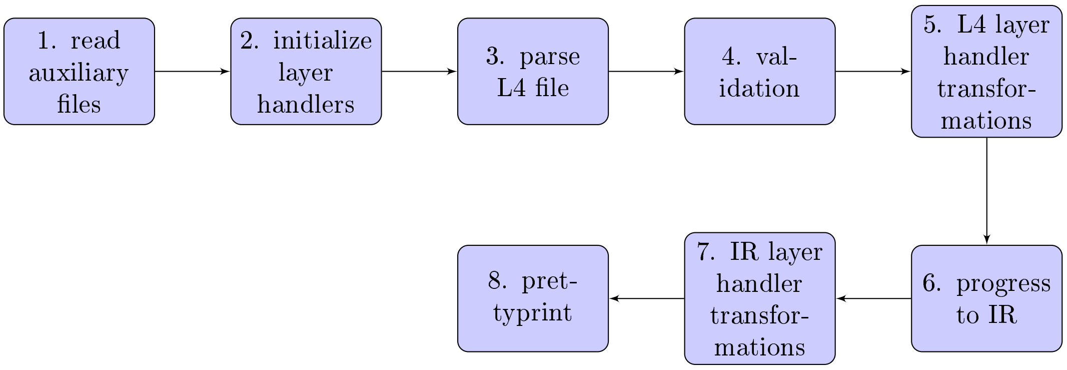

1. read auxiliary files
2. initialize layer handlers
3. parse L4 file
4. validation
5. L4 layer handler transformations
6. progress to IR
7. IR layer handler transformations
8. prettyprint

Figure 2.5: generation process

In step 1, knowledge, settings, and platform files are read in and parsed. The information contained is stored in collections, from which it can be retrieved by transformations. Initializing the layer handlers by setting up collections and data structures is step 2, the L4-program is parsed and the L4-AST is constructed in step 3. In step 4, the AST is reprinted and parsed to detect errors of the parsing process. Step 5 is the stage for L4-layer-specific transformations. Here only some of them are named for illustration. Integrating objects into corresponding collections and resolving object accesses. With step 6 the L4 tree is transformed into a tree of IR-nodes, where every node calls its progress-function and becomes an IR-node. Step 7, the execution of the IR-transformations by the `IR_LayerHandler` is the phase where most modifications of the AST, optimizations, and setups take place. For instance, parallelization statements are inserted into the AST and low-level optimizations like vectorization and loop unrolling are applied. Also special IR-nodes, that implement the *Expandable* trait, are *expanded* to multiple nodes. Nodes extending this trait can have different functionality and aims but are united in the property, that the point in time of their resolve is not important. This way, instead of letting multiple specialized strategies search for each of their nodes to resolve, all of them can be resolved at once by searching for `IR_Expandables`. Generally speaking, many transformations with various unrelated targets are executed here. The last step is emitting C++-code, by calling the *prettyprint*-function of every node. This function prints the representation of the node in a C++-program as a string to a target file, for instance a `for`-loop-node becomes the string `"for"` followed by the prettyprints of its loop-variable and body-statements in brackets [11, ch. 4][12]. More detailed information about ExaStencils workflow to generate a C++-Solver as well as about the different layers of ExaSlang can be found in [11] and [12].

ExaStencils Type Hierarchy

The generator is written in Scala, a functional and object-oriented general-purpose programming language. One of the Scala's special features is pattern matching, that is comparing not only values but also types and structures of variables via so called *case classes*. Nodes in the ASTs of ExaStencils are mostly Scala case-classes. This makes it easy to formulate transformations that search for certain types of nodes in the AST by pattern matching, extract attributes, and process them. This is illustrated in listing 2.4.

```
case IR_Assignment(dest, src, "+=") =>
          IR_Assignment(dest, IR_Addition(dest, src), "=")
```

Listing 2.4: pattern matching and replacing of an assignment

In this example, an `IR_Assignment` is a case-class. If the operand of this pattern-match-case matches the displayed pattern, two of its three attributes are extracted to the temporary variables

dest and src. Then a new case class is set up with modified values. dest += src becomes dest = dest + src (modified from listing 2.2. in [11, ch. 2]).
Generally, all nodes composing the AST of a layer are part of a class hierarchy. At the top of this hierarchy stands an abstract trait from which all other nodes inherit an attribute that will localize them in the source code, as well as make them annotatable. In this way annotations can be attached to and detached from any node. Then the class hierarchy starts resembling the conceptual hierarchy of ExaStencils layered system by splitting up to LX_Node and IR_Node, from which all nodes of the corresponding layer inherit. Even lower, for each LX_Node or IR_Node, nodes representing language constructs extend bit by bit. They are followed by traits for statements and expressions, then case classes for assignments, declarations, numerical literals and others [3, class hierarchy mode in IntelliJ].
Nodes, that are to be printed to C++ or for debugging reasons inherit the prettyprint-method from the trait PrettyPrintable. In this way the printing can be done in one traversal, matching on all nodes implementing PrettyPrintable and calling their corresponding method. Other interfaces and strategies with the same pattern, which is aiming at a one-traversal-execute-all, also exist. The process of converting the AST of one Layer to the next lower layer AST is one of them. This is the purpose of the LX_Progressable-traits [11, ch. 4]. Listing 2.5 shows an example of a progress method [11].

```
override def progress = ProgressLocation(
        IR_Assignment(dest.progress, src.progress, op)
)
```

Listing 2.5: L4_Assignment's progress method produces an IR_Assignment by progressing its attributes and source location

Selected Nodes

To give a summary of important AST nodes within ExaStencils for this work:

- L4_Statement and L4_Expression as well as their IR-counterparts resemble the two basic language supertypes
- An access to a plain variable is modeled as L4_VariableAccess that references its accessed variable. It progresses to a IR_VariableAccess
- A function call follows the same progress pattern and IR_FunctionCall holds a reference with the name of the function call, its return value datatype and a buffer of expressions that represent the function arguments
- When a multidimensional object is accessed at some entry an IR_HighDimAccess composed of a base and indices is constructed. The base resembles the accessed object and the indices specify the position of the entry to be accessed
- Datatypes generally implement the L4_Datatype trait and correspondingly in the IR, IR_Datatype
- L4_FieldAccess is produced, if a field is accessed. It holds a reference to the accessed field like a L4_VariableAccess does. Moreover, offsets of such accesses among some other modifiers for the field-access is saved here
- Equations, composed of a right and left side, are present as IR_Equation with buffers of expression for both sides

[3]

2.3.2 Matrices in ExaStencils

The Matrix Type

Matrices are a built-in type in L4 of ExaSlang. They can be used as a datatype of plain variables and fields. To specify a matrix variable, the user has to provide the inner datatype of the matrix, as well as its dimensions. Listing 2.6 displays how a matrix with inner datatype Double and two rows and columns is declared.

```
Var m : Matrix<Double , 2, 2> = {{1 ,2},{3 ,4}}
```

Listing 2.6: declaration and initialization of a matrix variable on layer 4

The L4 Lexer recognizes keywords like `Var` and produces a tokenstream, from which the L4-parser constructs the L4-AST. After transitioning to IR, the present node structure of the code line in listing 2.6 is depicted in figure 2.6. Some attributes like source location are left out.

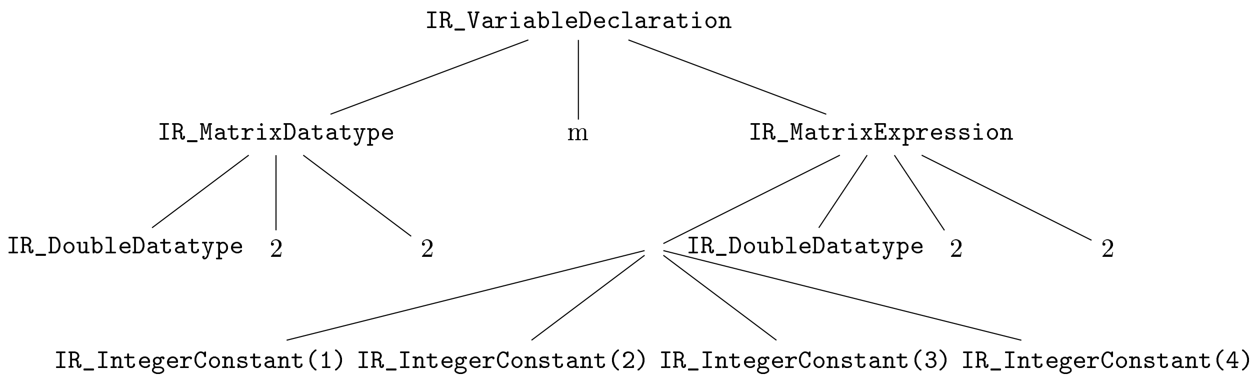


Figure 2.6: AST structure of a matrix variable declaration

Here it becomes clear, that matrices are represented in two different ways within the generator: firstly as an expression, implementing the expression trait: `L4_MatrixExpression` and `IR_MatrixExpression` both resemble an expression within a statement, consisting mainly of an inner datatype, the matrix dimensions, and the entries. Matrix entries are a collection of expression nodes of the corresponding layer in row-major order. The matrix expression classes have a getter and setter to modify single elements. The information on the right side of the declaration in listing 2.6 will be used to construct a matrix expression on layer 4, which becomes the right subtree of figure 2.6 after progressing. Secondly, there is the matrix data type, implementing the generalized datatype trait of the layer. This expression only describes the form of a matrix variable, which is its dimensions and inner datatype. It is present as the left subtree of figure 2.6. As a side note: an access to a matrix in code lines like:

```
Var m : Matrix<Double , 2, 2> = {{1 ,2},{3 ,4}}
m = 4
```

Listing 2.7: access to a matrix variable on layer 4

would is an `IR_VariableAccess`.
ExaStencils also offers different built-in functions for matrices, namely: access to elements, rows, and columns, dot product, cross product, transposition, and inversion [7].

Matrix Transformations

In compilers and code generators, *transformations* are operations that search for certain nodes or node combinations in an AST and modify, remove, replace, or add nodes to the AST. In ExaStencils, transformations match nodes by scala case-classes while traversing the AST. If a match is successful, the function embedded in the transformations, which defines the modification, replacement, or addition to the AST, is executed. *Strategies* are a collection of transformation, which are related by an identical aim. Executing a strategy means the ordered execution of the contained transformations [16]. Various transformations have to be applied on IR to bring matrices to a form, where they can be printed to correct C++-code. Additionally, the built-in methods have to be resolved. Furthermore, matrix expressions are used within the generation process without being declared on L4.
Matrix objects of the L4-AST are not modified by transformations on L4 yet, but only by transformations in the IR. What follows is a listing of the important modifications to the IR-AST related to matrices.
The transformations are grouped into three types: at first, preparatory, one time transformations

traverse the AST. They resolve, among other things, self-assignment multiplication of matrices e.g. $M = M * M$, to avoid errors by executing a matrix multiplication in place. Also, multiple function calls in one statement are extracted to separate variables, taking precaution against unwanted side effects of those function calls. Furthermore, all accesses to matrix variables are transformed into matrix expression, that consists of single access-nodes referencing certain entries of the matrix variable. This simplifies all strategies as there are fewer type checks to be done, because matrices thereon only exist as expressions and not as accesses anymore.
After some other minor modifications, the second group is executed. These strategies traverse the AST not only once, but repeatedly until they do not find any node matching their patterns anymore. They are called *iterative* in the following. Their task is resolving built-in functions. They have to traverse the AST multiple times, because in some cases, e.g. a transpose call can not be resolved on the first visit if the argument is a more complicated expression than just a variable access. If the argument is for instance a sum of two matrices, that sum has to be calculated first to be able to access single elements of the argument, which is necessary for transposing it. Otherwise, the transformation for the transpose call only sees an addition, which has no entries to interchange. The argument is then *not evaluatable*. To achieve correct resolving of such cases, another strategy, which resolves sums and multiplications of matrices, and the strategy to resolve built-in methods are executed alternately. This involves, that the latter ignores calls, whose arguments are not *evaluatable* yet. Being evaluatable means being a variable access or a matrix expression. Then the built-in functions can be executed. When no nodes match anymore, the iterative process terminates.
The third group maps assignments of matrices to C++ standard library function calls, such as $M = 1$ to std::fill, and linearizes multidimensional accesses to matrix variables. Furthermore, matrices in arguments of function declarations are transformed to implement pass-by-reference instead of pass-by-value [5].

2.3.3 Selected Topics of ExaStencils

Parsing with Scala Combinators

In section Parsing top-down parsing was described. The ExaStencils parsers are of the type *combinatory parser*, which is a subcategory of top-down parsers. Its characteristic trait is that a complete parser for a language is composed of many small parsers. Each of them parse separate language constructs and are combined via *combinators*. One smaller parser in a combinatory parser implements the parsing of one simple non-terminal. For more complex non-terminals like statements, the parser already consists of many combinator-connected, smaller parsers.
A combinatory parser is defined as a function that processes the input to a result. It matches an input on combinations of keywords, delimiters, and optionally other non-terminals. The same way one syntactic element of a language can be composed of many smaller syntactic elements, its parser is a composition of parsers of smaller syntactic elements. There exist various ways to combine parsers and build a more complex parser. The most important combinators are *sequencing* and *alternation*. Sequencing, denoted as ~, combines two parsers in order of input reading. So p1 ~ p2 means, that at first parser p1 tries to parse the input and if it succeeds p2 tries to parse the leftover input, which was not parsed by p1. Hence, a parser built by this combination applies p1 and p2 to the input. Alternation, written down as p1 | p2, tries to parse the input with p1 and returns its result if successful. If not, the same input is handed to p2 and p2's parsing result is returned. In ExaStencils another version of this combinator is used, that is |||. It essentially does the same as |, but if both component parsers succeed, the result of the one which consumed more input is returned. Up to now, it was never described how exactly the AST is built from a successful parse of input. In the case of scala combinator parsing, here the function application combinator (^^) is important. This operator tries to apply a given parser on input and afterwards applies a given function on the result of the parsing if it was successful [13, ch. 31].
In ExaStencils, the applied function is an assignment of parsing results to temporary variables by pattern matching and the consecutive construction of AST nodes. To visualize these concepts figure 2.7 is given.
In the *assignment*-parser, small grain parsers are combined via sequencing. One of them is *access*. An access may be just an identifier of the object to be accessed, so the identifier-parser is applied within the *access*. It is at the lowest level of parser granularity, and only checks if the current input

assignment =
(*access*~ ("=" ||| "+=") ~ *expr*) ^^ { case id ~ op ~ exp => new
IR_Assignment(id,op,exp) }

Figure 2.7: An overview of ExaStencils parsing

is of a certain token type, e.g. identifiers or numerical literals. These smallest parsers and delimiter strings like "{", "(", "=", ... compose the whole parser of ExaSlang. The *assignment*-parser tries to parse the input with the *access*-parser, followed by either "=" or "+="-delimiters and furthermore with the expression-parser *expr*. After an *assignments* body successfully parses, the function displayed on the right side of the function application combinator is applied to the parsing results. The results are a sequence of the results of *access*, an operator and the result of the *expr*-parser. A pattern match on the sequence is applied, assigning each part of it to the variables `id`, `op` and `exp`. Using the matched parsing results as child nodes, a new node is constructed. These results are again nodes constructed by the parsers *access* and *expr*. The assignment-node is afterwards incorporated into the IR-AST [8].
The ExaSlang 4 representation of new language constructs implemented in this work are implemented by Scala combinator-parsers.

Symbolic Operation Example

To demonstrate the process of: (i) resolving a function call of a built-in function formulated in the source program and (ii) executing a more complex method at compiletime, as well as (iii) executing a symbolic operation on a matrix, the matrix transposition at compile time is inspected in detail here. Given the declaration of listing 2.6 followed by a line like:

```
Var m_trans : Matrix<Double, 2, 2> = transpose(m)
```

Listing 2.8: example for a L4 function call

After lexing and parsing, following the pipeline of figure 2.5, the call arrives as an `IR_FunctionCall`-node within the IR-AST. A transformation for finding and resolving it is displayed in listing 2.9.

```
case call : IR_FunctionCall if call.name == "transpose" =>
// ... setup of transposed matrix expression
```

Listing 2.9: exmple of a transformation resolving an `IR_FunctionCall`

The transformation matches on `IR_FunctionCall`-nodes with the name "transpose" and replaces them with an `IR_MatrixExpression`, storing the result of the transposition of the function-call's argument. After building an `IR_MatrixExpression` of accesses to replace the argument of the transpose call, the corresponding sub tree of the right hand side in listing 2.8 looks as depicted in figure 2.8 (with simplification of the resulting higher dimensional accesses of the argument) [5].
To transpose the matrix, one needs to switch the expressions of the argument matrix expression at position (0,1) and (1,0). Note that the transformation can not know which actual values are at position (1,0) and (0,1), but only sees variable accesses. It has to *symbolically* interchange the entries. This will become important for symbolic inversions. Listing 2.10 shows the resulting C++ code after the transformation.

```
__matrix_double_2_2_t m_trans {m[0], m[2], m[1], m[3]};
```

Listing 2.10: result of transpose call with interchanged entries

Symbolic/Compile-time inversion

A symbolic inversion is executed at compile-time by taking entries from a `IR_MatrixExpression` and "forming" calculations. The term forming is used, because the calculation is not actually executed, but the argument entries are connected with new nodes resembling additions, multiplications, subtractions and divisions. They yield the result of the inversion when evaluated at run-time of the generated program. An illustration is given in listing 2.12, which shows the C++ result of an

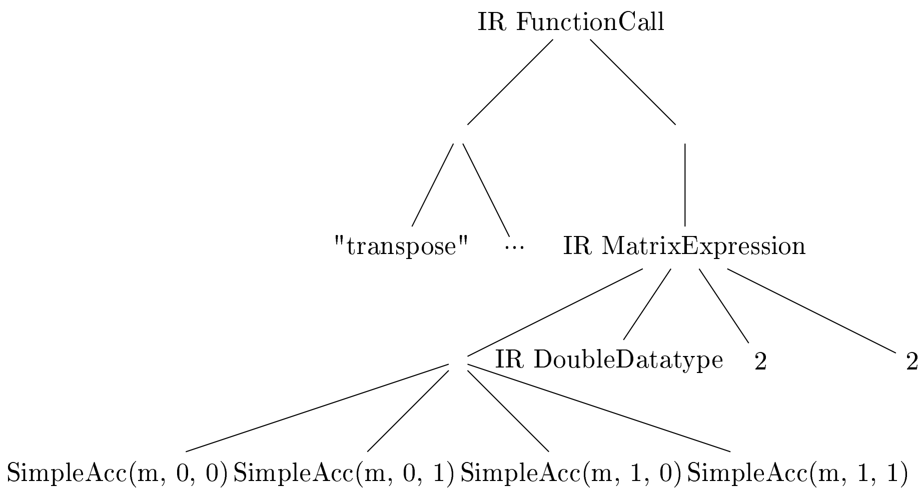


Figure 2.8: AST structure of a transpose call with expanded argument matrix

inversion call analogously to listing 2.10 after application of Cramer's rule for inversion at compile time. Within the generator, setting up the calculation is done as displayed in listing 2.11.

```
val a = m.get(0, 0)
val b = m.get(0, 1)
val c = m.get(1, 0)
val d = m.get(1, 1)
val det : IR_Expression = IR_Division(
          IR_RealConstant(1.0),
          (Duplicate(a) * Duplicate(d)) - (Duplicate(b) * Duplicate(c))
)
IR_MatrixExpression(that.innerDatatype, 2, 2, Array(
          Duplicate(det) * Duplicate(d), ...)
```

Listing 2.11: abstract of a compile-time operation within the generator

In lines 1-4, the entry expression of the input matrix `m` are extracted. Then, an arithmetic expression to calculate the determinant is set up in lines 5-8 by connecting the entries of the argument matrix with division, addition, and multiplication nodes (connecting nodes by * or - is an operator-overload which constructs `IR_Multiplication/Subtraction`-nodes). This is followed by building and returning the result expression in line 9. `Duplicate` expresses a copy operation. Similar to the operation for transposing a matrix, the exact values of the argument matrix are not known at compile time.

```
__matrix_double_2_2_t m_inv {
          ((1.0/((m[0]*m[3])-(m[1]*m[2])))*m[3]),
          // other entries follow ...
}
```

Listing 2.12: resulting entry of a inversion: calculation with references of argument entries

The first entry of the inverse, built in line 9 of listing 2.11, can be recognized in line 2 of listing 2.12 [5].
As every step in the calculation and every access brings another node object to the generation process, for larger matrices a symbolic inversion becomes more and more expensive. This is one of the problems motivating this work and will be discussed in detail in section Specialized Inversions.

Constant expressions

The matrix expression set up in line 9 of listing 2.11 only consists of *constant expressions*, i.e. expressions, which are constant at compile-time and not depending on other variables of the program. The opposite, a non-constant expression is e.g. a variable access, for which the exact value is not known at compile-time. Parts or whole expressions only consisting of constant expressions can be simplified by evaluating the expression's exact value and replacing it with another constant expression, only holding the evaluation's result. E.g. an `IR_Addition(IR_IntegerConstant(1), IR_IntegerConstant(2))` is simplified to `IR_IntegerConstant(3)`. The evaluation is basically the same as calculating the result at compile-time. In ExaStencils, there is a strategy, `IR_GeneralSimplify`, which traverses the AST and simplifies constant expressions to their evaluated result for some cases. This can reduce the AST-size after calculations at compile-time like inversion drastically. Besides, there exists another method, `IR_SimplifyExpression` to simplify a single expression, not the whole AST. It works by completely rebuilding an expression, evaluating, and summarizing constant expressions, in contrast to `IR_GeneralSimplify`, which applies a series of predefined simplifications like fusing double-negatives. This is more expensive but can lead to a stronger reduction of expression length. If a variable access is encountered while `IR_SimplifyExpression` tries to simplify an expression, the simplification is aborted [4], [6].

Collectors

In some cases, for a transformation more information from the program is necessary than is available locally at one visited node. For instance, checks and operations depending on the control flow of the program, which is not directly represented in an AST.
For a transformation, which checks variable accesses for the existence of their referenced variables, visiting a variable access with some identifier and type in the AST is not sufficient. The information, whether a variable with this identifier exists at this point in the source program is needed to validate the access. Here *collectors* come in handy. They are sort of a companion to a transformation, collecting information during the traversal of the AST. They essentially consist of two methods, *enter* and *leave*, which are called whenever a node is visited by a transformation and when it is left again. In each method, the entered or left node is checked for its type, and depending on the check, information is collected from the node. Furthermore, they possess a kind of buffer to hold the collected information. A programmer can then define which action should be executed when visiting and leaving a node.
Figure 2.9 explains visiting and leaving visually during the AST-traversal for a scope node, which is just a new code block delimited by { and }.

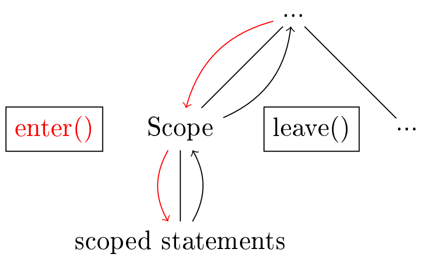


Figure 2.9: enter and leave during traversal via DFS

The scope node is entered on the way down and left when all its children were traversed.
To give an example for the use of collectors, a *variable declaration collector* is explained in detail here. Its task is to collect variables when they are met in the AST, as well as manage them according to the scope they were found in. To do that, the collector holds a list of lists, whereby the inner lists contain variable declarations and each entry of the outer list resembles a scope. For traversing the AST for a code piece like in listing 2.13 the resulting datastructure is illustrated in figure 2.10.

```
int a;
{
        int b;
}
b = 1;
```

Listing 2.13: code piece with a scope

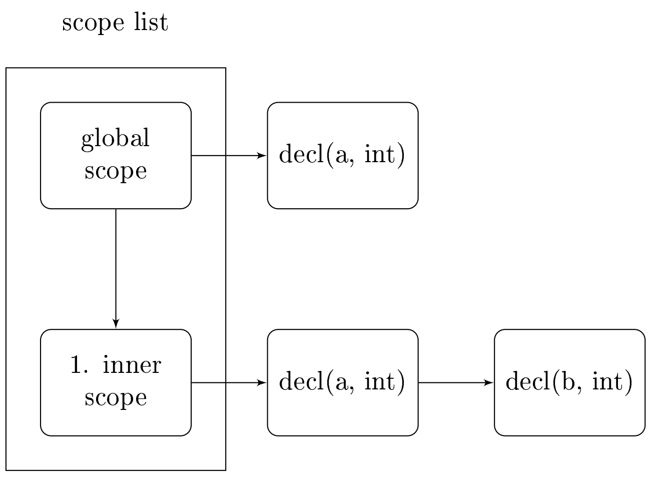


Figure 2.10: Collector data structure at line 4

The data structure is constructed as follows: every time the traversing transformation meets a node, the collector's enter method is called. This method has two modes depending on the type of node: a variable declaration is added to the first entry of the scope list, that is the current scope. If the node opens a new scope, e.g. by directly declaring a new code block as in the example, the first entry of the scope list, which contains the variables of the surrounding scope, is copied and added to the head of the list. This corresponds to the fact, that in the newly opened inner scope all variables of the surrounding scope exist as well. If all children of the scope opening node are traversed, the transformation leaves the scope node, calling the collectors leave-method. As variables of lines 2-4 are not available anymore after line 4, the first entry of the scope list is removed. This leads to the scope list effectively working as a stack, where opened scopes are pushed and left scopes are popped. In this way, an accurate tracking of variable availability is achieved. When visiting the assignment of the variable `b` in line 5 in the source code, the transformation can identify the access of `b` as illegal and produce an error, as there exists no declaration of a variable `b` in the current(global) scope [11, ch. 2], [9].
Collectors are used for the tasks in section Matrix Shape Classification, to check if there exist writes to certain variable declarations up to the current point of traversal, as well as collecting variable declarations to extract their initial expressions.

Fields and `solve locally`

One of the most important constructs of L4 is *fields*. They are mostly defined in the PDE domain, storing values per grid point. All calculations to obtain a solution for a PDE problem are done via fields. Simple iterative solvers can easily be set up by defining updates of field entries within a field loop. Listing 2.14 displays a Gauss-Seidel update on the solution field `u`.

```
loop over u {
        u += 1.0 / diag(Laplace)  * (RHS - Laplace * u)
}
```

Listing 2.14: Gauss-Seidel update with a field loop

(modified from listing 3.27 of [11, ch. 3]) Inside the loop, `u` denotes the current entry of the field iteration. `Laplace` is a stencil representing the Laplace differential operator and `RHS` is the right hand side of the differential equation, also given as a field. The loop over `u`, too accesses `rhs` at the corresponding position to `u`, assuming they are defined in the same domain. In the generation process, these accesses are represented as `L4_FieldAccess`. The user can optionally instruct the generator to allocate multiple copies of a field to make a *Jacobi-type* update of the field possible. Different copies of the field are then accessed via different *slots*.
ExaSlang additionally provides a way to solve for multiple entries of unknown fields through one statement. This is done by setting up a small linear system of equations for chosen entries of the solution field and then obtaining a solution for these entries by solving the system. Within a `solve`

`locally`-statement, the user can provide equations which should be full filled for a field access. The local system is composed of the provided equations. Typically, multiple field accesses with offsets are subject to multiple different equations. An example for the `solve locally`-statement is given is listing 2.15.

```
loop over u {
        solve locally {
                u@[0, 0] => laplace@[0, 0] * u@[0, 0] == rhs@[0, 0]
                u@[1, 0] => laplace@[1, 0] * u@[1, 0] == rhs@[1, 0]
        }
}
```

Listing 2.15: `solve locally` for two positions of the field `u`

(modified from listing 3.29 of [11, ch. 3]) The generator is told that the current point of iteration is subject to the equation on the right of the arrow in line 3 and a shifted iteration point is subject to the equation in line 4. The access to `u` can be shifted by offsets, in this case by one grid point in y direction for the equation in line 4. The generator sorts the given equations with respect to an unknown entry, i.e. `u`. When all of those are on the left side, the local linear system can be set up. In the example, this is a 2×2 diagonal system matrix, a 2×1 unknowns vector and 2×1 right-hand-side vector. To do this, local matrix variables for unknowns, right hand side and system matrix are declared and statements to fill them at run-time are prepared. This is a case of the usage of matrices without them being present explicitly in the L4-program. After solving the system by inversion, following $u = A^{-1} * f$, the results are written from the local unknowns back to the global unknowns `u`. Depending on the size of the system to solve and an attribute in a knowledge-file, a symbolic or an inversion at run-time are employed. `solve locally` also offers options to exert damping on the equations with a damping parameter and writing the results to a different slot of `u` as a Jacobi-type update. Furthermore, for loops with identical system matrices for each iteration, the calculation of the inverse system-matrix can be drawn before the loop to avoid multiple repetitive calculations. Then, in the loop body, only a matrix-vector-multiplication is executed to solve a local system [11, ch. 3].
Moreover, if the user knows about the shape of the system matrix and latter has a Schur-shape as given in definition Schur complement with an A-block containing multiple zero-blocks, the generator can exploit this by applying a specialized algorithm. Such shapes either arise by choosing the equations or unknowns or their localization in a way aiming to build a system with such a shape or in certain applications. One of them are the Stokes equations. These are given by the momentum balance and the continuity equation for two dimensions, whereby the kinematic viscosity is set to 1:

$$-\nabla^2 * u + \frac{\partial p}{\partial x} = f_u \tag{2.7}$$

$$-\nabla^2 * v + \frac{\partial p}{\partial y} = f_v \tag{2.8}$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{2.9}$$

Written as a block system:

$$\begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix} * \begin{pmatrix} \mathbf{u} \\ p \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ 0 \end{pmatrix}$$

with $B = -\nabla$ and $A = -\nabla^2$. Simple point-wise solvers use the inverted diagonal entry of the system, which is zero for the pressure-part, as can be seen in equation 2.10. This is one reason why they are not applicable as smoother here. Constructing small local linear systems with a combination of pressure equation of one grid point and velocity equations of surrounding staggered grid points mitigates this problem. In 2D cases, the local system is then structured as follows:

$$\begin{pmatrix} A_{11} & 0 & B_1 \\ 0 & A_{22} & B_2 \\ C_1 & C_2 & D \end{pmatrix} * \begin{pmatrix} U_1 \\ U_2 \\ V \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \\ G \end{pmatrix} \tag{2.10}$$

The representation of the system matrix is called *Schur complement.* Every row in this block system stems from an equation in a *solve locally*-statement, defined for an entry of the iterated field in the neighbourhood of the current point of iteration, as previously. Without setting up an inverse matrix, the system can be solved by decomposing it into its blocks A_{11}, A_{22} ... etc. and computing the Schur complement matrix S:

$$S = D - C_1 * A_{11}^{-1} * B_1 - C_2 * A_{22}^{-1} * B_2 \tag{2.11}$$

It is identical to definition 2.1, where the second term of the right side is decomposed to the separate parts of the inverted blockdiagonal matrix A and the sub matrices B and C of M. 2.11 is calculated to solve

$$S * V = G - C_1 * A_{11}^{-1} * F_1 - C_2 * A_{22}^{-1} * F_2 \tag{2.12}$$

for V. 2.12 arises by interchanging the first block equation of system 2.10 for U_1, and the second block equation for U_2 and inserting them into the third block equation. After calculation of V, the remaining unknowns of U_1 and U_2 can be calculated by solving the first and second block equation:

$$U_1 = A_{11}^{-1} * (F_1 - B_1 * V) \tag{2.13}$$

$$U_2 = A_{22}^{-1} * (F_2 - B_2 * V) \tag{2.14}$$

After the procedure, U_1, U_2 and V are written to the solution fields from which the system is set up.
In the previous version of the generator, the decomposition is implemented only for certain cases in terms of blocksize, which are 5×5 matrices with two 2×2 blocks on the diagonal and 7×7 matrices with three 2×2 blocks on the diagonal, where m, the size of the matrix block D, is one [11, ch. 6].
The decomposition is extended and used in the work described in section Solving Local Linear Systems.

3 Implementation

3.1 Overview

The implementation part of this work is composed of components with different motivations and aims, which are labelled as *tasks*.
In the following sections, the version of the generator present prior to this work is described. Then, the methods used to implement and test each task are summarized. The rest of the section consists of a detailed description of the motivation for each task and its implementation. In the end of the Implementation-chapter, a short list of commands that activate the extensions is given.
In the following, *compile-time* denotes the time of the generator generating the application, *run-time* is used to name the time, the generated application needs to run, and *compile-time of the application* for the compilation process of the C++-compiler.
All tasks are related to matrices: beginning with (i) extension of the user interface of matrices, that is the ExaSlang 4 language, next to (ii) the refactoring of the general processing pipeline of all kinds of matrix variables and operations within the generator and ending with (iii) the extension of methods to optimized especially computation-expensive matrix operations.
The tasks are numbered as follows:

1. Dedicated Nodes
2. Slicing and Bracket Access
3. Matrix Shape Classification
4. Specialized Inversions
5. Solving Local Linear Systems

In the first task, processing of matrix datatypes and operations present as IR-AST is treated. It consists of a refactoring of the AST-strategies described in section Matrix Transformations. They implement the modifications and additions to the IR-AST related to matrices. The refactoring is done by using a similar pattern as used in the expansion of `Expandables`: similar nodes are resolved by letting them inherit from one trait and then targeting the trait in transformations. To make this possible, dedicated AST-nodes for matrix operations are set up.
Task two extends the ExaSlang 4 language by slicing and access operations for matrices.
Thirdly, the automatic classification of matrix shapes is presented. Before inverting or solving a small linear system of equations, the matrix of the problem can be inspected for a present shape. Then, a fitting algorithm to solve the problem, which exploits the matrix shape, can be applied. Additionally, the option to provide matrix shapes with inversion calls, field declarations, and matrix declarations in L4-programs is described, as well as the representation of matrix shapes as a node in the AST. The implementation also aims at making the extension with additional shapes easy.
Algorithms to invert a matrix under exploitation of zero entries in it, that define its shape, are implemented in task four. They are realized for compile-time and run-time execution. For run-time, this only leads to reduced run-time of the inversion. For compile-time execution, it also reduces the AST size. The production of large expressions when inverting at compile-time, increasing the size of the AST by arithmetic operator nodes, limit the compile-time inversion in size of the matrices to invert. Two types that appear often are targeted: Schur-shapes and block-diagonal shapes.
Task five is the introduction of a new construct, the `SolveMatrixSystem`-class. Its task is to solve small linear systems of equations directly, without calculating an inverse matrix. It is used within `solve locally`-statements in the compilation process. It also has a representation on layer 4, so it can be used directly, too. Next to solution of general systems, it implements specialized algorithms to solve small linear systems of equations with the previously mentioned matrix-shapes.

3.1.1 Prior State of the Generator

A description of the starting points for every task is given here.

1. The strategies for resolving matrix operations, described in section Matrix Transformations, are implemented by picking `IR_FunctionCall`-nodes from the IR-AST with corresponding names, like "transpose". The result of an operation on matrices is calculated directly in the strategy by processing the `IR_Expression`-arguments of the call. This leads to a long series of matches on function calls and the calculation of results of the call on unspecialized, general AST-nodes like `IR_FunctionCall` and `IR_Expression`. Lots of casts from one AST-node to another occur. This is necessary to be able to operate with the required attributes, e.g. an argument of an inverse-call has to be transformed to an `IR_MatrixExpression` to retrieve its size. The strategies match on every `IR_FunctionCall` and check its name to filter the targeted types of calls.
   Furthermore, only inverse-calls can be resolved at run-time. A single, point-wise mechanism operating in different strategies resolves inverse-calls in a way so that correct C++-code is generated. Point-wise in the sense that it is only and specially targeted on inverse-calls. Only with difficulty it can be extended to execution of other operations at run-time. Therefore, for other operations to be executed at run-time, most likely more point-wise mechanisms would have to be implemented, continually making the resolve strategies more complicated.

2. Access to a matrix is only possible by a function. Slicing also is implemented only as a function and restricted to retrieving or setting a single row or column of a matrix.

3. Classification of matrix shapes and their use in inversions are not implemented. Only for two single cases of Schur-matrices, zero blocks can be exploited in `solve locally`-statements by decomposition of the system matrix. This is the subject of section Fields and `solve locally`. A linear system of size 7 within a `solve locally`-statement can be solved, if the system matrix has Schur-shape with blocksize of $A = 2$. The generator has to be explicitly told that such a case is present by a knowledge attribute.

4. Inversion at compile-time is limited to matrices of size 5. When inverting one matrix of size 6, the compilation process already takes minutes. Small linear systems of equations are solved in `solve locally`-statements by inversion, so `solve locally`-statements have the same limit for cases other than the ones mentioned in point 3. Usage of shapes is not implemented for inversion.

5. The direct solution of linear systems of equations without calculating an inverse is not implemented.

3.1.2 Method

The approach to tackle each of the tasks consists of implementation and testing.
If the task has no way of addressing it on L4 yet, the L4-parser is extended accordingly. Task 4 is excluded from this. In order to execute the task in the IR, intermediate nodes on L4 have to be set up, transporting information to IR. This is done by adding one or multiple nodes as L4-nodes. They are constructed by the L4-parser after parsing an L4-program and inserted into the L4-AST. Then, a `progress`-method to map them to corresponding IR-nodes is implemented. Besides that, strategies to find the node in the IR-AST and to resolve it, are built. They usually call an internal method and replace found/matched nodes with the method's result. The internal method implements the actual functionality, such as calculating an inverse. In this way, strategies searching for their target nodes and calculation of the replacements are separated.
Errors in the implementation can occur at different points of the whole process from Layer 4 to execution of a C++-application:

- an error occurs while applying transformations to the AST, then the generation process is stopped
- the error is not fatal enough to stop the generation process, but incorrect C++-code is generated, which leads to the C++-compiler producing a compilation error
- a logical error is found by viewing and inspecting the generated C++-code

- the results of the application are checked for correctness

Tests for the implementation are done by checking for errors on those four occasions. The checks are done by defining an L4-program, generating the C++-application, compiling and running it, as well as comparing the results and viewing the generated code. The comparison happens via a built-in `compare`-method in L4, which takes two arguments and generates a comparison for the run-time of the application. Because a program can not start and check its own generation, compilation, execution, and comparison, a python script executes the tests and does the checking. Tests in this manner are implemented for all tasks. Additionally, example applications that are included in ExaStencils, such as a Stokes-problem, are run.

3.2 Dedicated Nodes

3.2.1 Motivation

Implementing dedicated nodes for a functionality means that instead of operating on a general node a node explicitly designed for the functionality is set up. It stores corresponding information about the functionality as attributes and can provide type safety. To use such nodes, the targeted, general node has to be replaced by the dedicated node at some point in the generation process. After the replacement, strategies only operate on the dedicated node. To illustrate this idea, listing 3.1 is given. It contains a part of the strategies to resolve function calls connected to matrices from the previous version of the generator.

```
case call : IR_FunctionCall if (
        call.name == "dotProduct" ||
        call.name == "dot"
) => ...
```

Listing 3.1: matching on `IR_FunctionCalls`

The transformation matches on `IR_FunctionCalls` whose name is "dot" or "dotProduct". The subsequent calculation has to bring the arguments of the call, which are of type `IR_Expression`, to matrix form. It also checks every `IR_FunctionCall` in the IR-AST for its name.
Operating instead on a dedicated node, as shown in the following listing:

```
case dp : IR_DotProduct => ...
```

Listing 3.2: matching on a specialized node

has three advantages:

- Searching for the targeted node in the AST is simplified after initial the replacement, because not all general nodes have to be matched to extract and check the value of some attribute.
- The code of strategies is more readable and shorter.
- The dedicated nodes can accomodate the specifics of the functionality, e.g. an attribute for a matrix shape in a dedicated node for inverse calls.

Indirectly, they also enable using traits to abstract AST-operations, which are executed for multiple nodes in the same or a similar way. One example for this is the *resolving* of all matrix operations (dot product, inverse, transpose etc.). It describes finding them in the AST, calculating the result and replacing the node for the operation with the result in the AST. The `Expandable`-nodes in ExaStencils follow this pattern, too.
Strategies, which execute the abstracted operation for all applicable nodes can be formulated a lot easier with such traits, as can be seen for expansion of all nodes inheriting from `IR_Expandable`:

```
case expandable : IR_Expandable => expandable.expand()
```

Listing 3.3: matching on an `Expandable`-node

The alternative is having many strategies, each searching for their target node and resolving it. This results in many AST-traversals or less strategies but more complicated matching, due to many different cases for different node types. This becomes apparent for the following strategy, which resolves matrix functions in the previous version of the generator:

```
case call : IR_FunctionCall if (
        call.name == "dotProduct" ||
        call.name == "dot"
) => // ... resolve dot product
case call : IR_FunctionCall if (
        call.name == "crossProduct"
) => // ... resolve cross product
case call : IR_FunctionCall if (
        call.name == "transpose"
) => // ... resolve transpose
// others follow
```

Listing 3.4: matching on `IR_FunctionCalls`

3.2.2 Matrix Function Nodes and Traits

In this work, for all built-in operations related to matrices, dedicated nodes are implemented. They consist of attributes for their arguments, a method to resolve them to a result and sometimes other specialized attributes needed for the matrix operation. Additionally, three traits are designed to abstract three AST-operations executed for many or all the matrix operations. The nodes are present in the IR and include: inversion, dot product, cross product, determinant, access and slicing.
The first trait implemented is called `IR_ExtractableMNode`, denoting extractable matrix-operation nodes. Nodes inheriting from it are extracted from statements to separate variable declarations. This is done by a strategy searching all statements of the AST for extractables and producing new declarations for them, also replacing them with variable accesses in the statement. It avoids unwanted side effects in statements with multiple chained function calls. Additionally, resolving chained function calls is simplified by this approach. Nodes extending this trait furthermore implement the method `isExtractable`. The extract strategy only moves them to separate declarations, if this method returns true. Thus, each node can decide, whether it is extracted or not and under which conditions depending on arguments. For instance, a slice of size 1×1 retrieved from a matrix can be inlined conveniently and without complications, while a slice of larger size, which also has to be sliced at runtime, cannot be inlined easily. Inspecting its arguments, a slice node calculates how large the slice is and decides whether it wants to be extracted from its statement or not. This way, extraction can be adapted easily, punctual and specialized for each operation, by extending the trait and implementing a policy in the `isExtractable`-method.
The second trait, `IR_ResolvableMNode`, abstracts the resolve operation for built-in matrix functions. It consists of two methods. `isResolvable` checks if a node inheriting from it is ready to be resolved. The other one, `resolve`, executes the nodes internal function, which implements the calculation of the result.

```
case class IR_DotProduct(
        var left : IR_Expression,
        var right : IR_Expression
) extends IR_ExtractableMNode with IR_ResolvableMNode {
override def resolve() : Output[IR_Expression] = {
/* internal method calculates dot product */
/* utility method transforms to IR_MatrixExpression */
        IR_CompiletimeMatOps.dotProduct(
                IR_MatNodeUtils.exprToMatExpr(left),
                IR_MatNodeUtils.exprToMatExpr(right)
        )
}
override def isResolvable() : Boolean = {
        /* utility method checks evaluatability */
        IR_MatNodeUtils.isEvaluatable(left) &&
        IR_MatNodeUtils.isEvaluatable(right)
}
override def isExtractable() : Boolean = true
}
```

Listing 3.5: The dot product node

IR_DotProduct should be extracted from statements, so it extends IR_ExtractableMNode. Since it has no restrictions in its extraction, its isExtractable-method always returns true. It can be resolved to a result expression, hence it extends IR_ResolvableMNode, inheriting its two methods. It also contains two attributes for the left and right arguments. It is always resolved at compile-time.
The third trait, IR_RuntimeMNode, is designed to enable convenient processing of matrix functions, which are executed at run-time or at compile-time. It replaces and extends applicability of the point-wise resolve mechanism of the inversion call in the old implementation.
For inversion, the decision whether to execute it at run-time or compile-time is made at the beginning of the compilation process by specifying a knowledge parameter. For other functions, it depends on the arguments. For those, the decision is made at compile-time, after checking the arguments for certain properties. E.g. for an inversion or eigenvalue calculation, the decision could be made based on the size of the input matrix, instead of setting it for all inversions via the knowledge file. For large matrices, run-time execution is chosen to avoid the problems described in section Specialized Inversions, otherwise compile-time is chosen to achieve shorter run-time of the application. Another example is slicing: if the size of a slice is not constant, but depending on a loop index, a matrix expression can not be constructed at compile-time, so the slicing can not be executed at compile-time. This is due to the fact that matrix expressions have to be constructed from constant integers defining their size, while the slice size is defined by variable access, which is a non-constant expression. In this case, the slicing has to take place at run-time.
The fact that these decisions can only be made at some point in the generation process depending on their arguments made the approach reasonable. The iterative resolving of matrix functions also makes this more complicated.
For such cases the following approach is implemented:

1. replace the function call by an intermediate IR_RuntimeMNode, which implements the inherited method resolveAtRuntime. The decision about run-time and compile-time execution is made by each node individually in this method. It returns a boolean value.

2. a strategy progresses all IR_RuntimeMNodes to a compile-time or run-time version of it depending on the return value of resolveAtRuntime. These nodes both implement the IR_ResolvableMNode trait. The resolve method of each implements the function, e.g. inverting the matrix or producing a slice of a matrix, for compile-time or for run-time.

3. another strategy resolves all IR_ResolvableMNodes

The complete processing of all traits and including the function nodes is described in section Resolve Process.
When new functions for matrices are added to the framework, they can also be set up as dedicated nodes. They can implement their own decision about execution at run or compile-time in a resolveAtRuntime method, and are automatically extracted and resolved by the implemented strategies, as they inherit from the other traits. Next to bringing order into the run/compie-time-execution-decision, it also offers an easy way to implement new built-in functions to be resolved at different points in time: inheriting from the traits.

Division of compile-time and run-time execution nodes

The splitting to resolvable nodes for run-time and compile-time in step 2 has the following reason: to generate a function for run-time, the destination variable of the result has to be known. This means that they have to be resolved with their assigned destination as one statement. Compile-time methods can be resolved independent of their destination variable, as they are mapped to expressions in C++. Consequently, the run-time version of the node is an IR_Statement and the compile-time-version is an IR_Expression. Furthermore, the run-time node replaces the whole statement, also retrieving which variable the result to assign to, whereas the compile-time node only replaces the intermediate node, which is also an IR_Expression. This difference is illustrated

by the following example of slicing at run-time and compile-time. The displayed slice-functions are implemented in this work and described in detail in the next section.
Suppose the layer 4 code line:

```
Var slice : Matrix<Real, 2, 2> = getSlice(mat, 0, 0, 2, 2)
```

Listing 3.6: a getSlice call on L4

`getSlice` has the signature `getSlice(matrix, row offset, column offset, row width, column width)` and slices a stripe or a submatrix from another matrix. In the generated code and assuming `mat` is also a 2×2 matrix, the generated code after compile-time execution reads:

```
__matrix_double_2_2_t slice {mat[0], mat[1], mat[2], mat[3]};
```

Listing 3.7: Slicing resolved at compile-time

The slice expression in the initialization of `slice` does not depend on the target of the assignment. In the generation process, the assignment looks as follows, when split from the declaration:

```
IR_Assignment(SimpleAcc("slice"), IR_GetSlice(SimpleAcc("mat"),0,0,2,2))
```

Listing 3.8: the correspoding AST node for the slice example

As resolving the `IR_GetSlice`-node does not depend on the destination variable, it can be done independently of the rest of the assignment.
Slicing generated for run-time execution on the other hand is pictured in listing 3.9:

```
__matrix_double_2_2_t slice;
for (int i = 0; i<2; ++i) {
        for (int j = 0; j<2; ++j) {
                slice[((2*i)+j)] = mat[((2*i)+j)];
        }
}
```

Listing 3.9: Slicing resolved at run-time

Here, the assignment of the slice entries is done at run-time. To generate the loops, the destination of the slice assignment, which is `mat`, must be known for the access to it in line 4. Due to that, a node as in listing 3.8 has to be replaced as a whole by another statement node, which incorporates the destination access. The same reasoning applies for all matrix functions, which can be executed at run-time and compile-time. This is why compile-time and run-time matrix function nodes are split and also have different types, `IR_Expression` and `IR_Statement`.

Annotations for evaluatability and matrix operations

The `isResolvable`-method of a `IR_ResolvableMNode` checks, whether a function-call-node is ready to be resolved. It does that by inspecting the arguments for their type. If the argument is an arithmetic operation of matrices or another function call, it has to be resolved first. Then the argument is not evaluatable, `isResolvable` returns false and the node is resolved in another iteration of the strategy. If the argument is present as a variable access or matrix expression, it returns true. As a result, the internal functions can operate on the evaluatable argument.
Type checks also have to be done to make sure an arithmetic operation or a function call are related to matrices. Scalar additions are not to be resolved by the strategies of this work, as well as e.g. a dot product of two tensors.
To avoid repeated, expensive type checks, the results of the check, such as that an addition does contain a matrix in its operands and is therefore to be resolved, the results of a check are saved in form of an annotation. The next time this node is checked for its evaluatablility or if it resembles a matrix operation, the typecheck is left out, because the node has an annotation, marking it as evaluatable or as a matrix operation.

Algorithm 3.1 Resolve process of matrix functions

```
 1: replace function calls with dedicated (intermediate) function nodes
 2: if IR_ExtractableMNode.isExtractable then
 3:     extract IR_ExtractableMNode to separate variable
 4: end if
 5: while one of the following matched a node do
 6:     if IR_RuntimeMNode.resolveAtRuntime then
 7:         progress assignment with intermediate IR_RuntimeMNode to run-time version
 8:     else
 9:         progress intermediate IR_RuntimeMNode to compile-time version
10:     end if
11:     if IR_ResolvableMNode.isResolvable then
12:         IR_ResolvableMNode.resolve
13:     end if
14:     resolve arithmetic operators
15: end while
```

3.2.3 Resolve Process

The complete process to resolve matrix functions is formulated in algorithm 3.1. The operation in line 1 sets up the intermediate `IR_RuntimeMNode`, which are progressed to `IR_ResolvableMNodes` for the matrix functions that can be resolved at run-time and at compile-time. A default node like `IR_DotProduct` is set up for all other matrix functions. Then all nodes which are to be extracted are put to separate variable declarations and replaced with an access to it in the statement they were found in. Afterwards, the iterative process of resolving matrix functions starts, transforming matrix function nodes to run-time or compile-time execution nodes. Besides resolving them, arithmetic operators with matrix arguments are resolved.

3.3 Slicing and Bracket Access

3.3.1 Motivation

The possibilities to slice a matrix are extended. In the previous state of the generator, on L4, single entries or rows and columns of matrices can be retrieved by the methods `getRow`, `getColumn` and `getElement`. Besides, rows, columns and entries can be set with `setRow`, `setColumn` and `setElement`. The methods have limitations: rows or columns can not be set to scalar values, the methods can not retrieve or set submatrices but only single lines and calls can not be chained. E.g. `setColumn(matrix, 0, getColumn(another matrix, 1))` is not possible. Slicing matrices and accessing only parts of a matrix without such restrictions has a lot of possible use cases.
The new implementation consists of two parts:

1. the methods `getSlice` and `setSlice`, which are also able to slice and access/set sub matrices of a matrix.
2. set up of a new interface for slicing and access to matrices: the bracket-operator. Statements containing matrix variables and a bracket-operator are parsed and mapped to intermediate nodes, the `L4/IR_MatAccess`. This is done to unite slicing or access to different types of matrix variables, which are by default resolved very differently from each other in the generator. `IR_MatAccess`-objects are transformed to `getSlice` and `setSlice`, which already implement the functionality.

3.3.2 `getSlice` and `setSlice`

`getSlice` and `setSlice` extend the capability of `getRow/Column` and `setRow/Column` to whole sub matrices. Both are parsed as function calls. Furthermore, dedicated function nodes are constructed for them in the first step of algorithm 3.1. They are then resolved together and in the same manner as other matrix functions like inversion.
Setting a slice to a scalar value or another matrix of fitting size is done by specifying a call

of the form: `setSlice(dest, row offset, column offset, row width, column width, src)`, containing the parameters:

- `dest`, matrix in which the slice is to be set
- `row offset`, `column offset`, location of the slice in `dest`
- `row width`, `column width`, size of the slice
- `src`, input for the slice, can be a matrix or scalar value

The corresponding function node extends `IR_Statement`, as `setSlice` is used as a statement. It generates a nested loop when resolved, which sets the entries of `dest`.
`getSlice` has the same signature as `setSlice`, exept only the slice location, size and the matrix from which to slice is specified, whereas the parameter `src` is left out. It acts as an expression on right side of assignments or in function calls, so its function node extends `IR_Expression`. Listing 3.19 and 3.6 demonstrate their usage.

```
Var m : Matrix<Double, 3, 3> = {{1, 2, 3}, {4,5,6},{7,8,9}}
// retrieve the first two entries of the last row of m with getSlice
// and place them at the last two entries of the second row of m
setSlice(m,1,1,1,2,getSlice(m,2,0,1,2))
```

Listing 3.10: a chained call on L4

Slices of run-time determined size

A `getSlice`-call can be executed at run-time and compile-time, depending on its operands. In section Division of compile-time and run-time execution nodes, the differences between resolving a function for run-time and for compile-time are presented with the example of slicing. In the following, the reason to resolve functions at run-time besides long expressions stemming from calculations, as is the case for inversions, is given. Suppose a sub matrix of a larger matrix is sliced to e.g. invert it afterwards. Assume that the slice call has to be extracted to a separate declaration. To declare the declaration, the size of the slice must be known. This is necessary, because the separate declaration is of type matrix, with the same size as the slice. If the slice size depends on e.g. a file input, it is not determined at compile-time, so a statement like in listing 3.7 can not be generated. To give an example on L4:

```
Var runtime_dependent : Int = // ...
Var slice : Matrix<Double, 2, 2> = getSlice(m,0,0,runtime_dependent,2)
```

Listing 3.11: a getSlice call on L4

As the size of the slice is not known at compile-time, not only loops have to be generated to be able to set a number of entries that are only known at run-time, but also the memory of the slice can not be directly allocated from the stack. Therefore, heap memory of run-time determined size is allocated to store the declaration.

```
double* fct_getSlice_2;
fct_getSlice_2 = new double[(2*runtime_dependent)];
for (int i = 0; i<runtime_dependent; ++i) {
        for (int j = 0; j<2; ++j) {
                fct_getSlice_2[((2*i)+j)] = m2[((2*i)+j)];
        }
}
```

Listing 3.12: a getSlice call in C++

Having to declare variables of run-time determined size rarely occured in this work and is only implemented as an initial version. If the implementation of such mechanics is done, the following should be considered: to prevent prolonging calculations by such time-consuming system calls as allocating memory from the heap, they should be moved to the start of the program alongside with setup and initialization operations. Furthermore, the freeing of the heap-memory should be executed after all calculations at the end of the program.

3.3.3 Bracket Access

The second part of this task is implementing slicing and accesses to matrices with the bracket operator. The ExaSlang 4 forumlation looks as follows: Either only indeces or index ranges are defined in one or two brackets after a variable identifier. Each bracket corresponds to one dimension of the matrix. An index range consists of start index separated from end index by a double dot. Also, to take the whole range of one dimension of the matrix variable, only one double dot can be specified. If only one range is added in a bracket behind the variable identifier, the whole range of the other dimension is chosen by the generator. The operator can be used on the left side of an assignment, to access a submatrix of the variable, or on the right side of an assignment to retrieve a slice. The bracket operator is exemplified in the following.

```
Var mat : Matrix<Double, 2, 2> = {{1,2},{3,1}}
mat[0][0] = 2 // first entry set to 2
mat[0:2][0:1] = 3 // first column set to 3
Var slice : Matrix<Real, 1, 2> = mat[0][:] // first row retrieved
```

Listing 3.13: Accesses and slicing via bracket operator on L4

Resolving of Bracket Slicing and Accesses

The bracket-operator can be applied to different types of matrix variables. These are resolved independend of each other in very different ways, so bringing them together to resolve the bracket-operator for all of them at one place is not straight forward to implement. They are also not parsed as functions but as one of many other types of accesses and can occur on both sides on assignments. After parsing, bracket-operators are progressed to the intermediate nodes, `IR/L4_MatAccess`, in IR or L4, depending on the type of variable that is accesses or sliced. Then, they are mapped to the slice methods of the last section, `getSlice`, `setSlice`, `getElement` and `setElement`, as they implement the same functionality, which then become loops are matrix expressions in C++.
The resolve process of the bracket operator begins at the L4-parser. The parser for a generic access, which parses accesses for all types of objects, is extended by the new parser for the brackets, called `matIndex`:

```
matIndex = (index ||| rangeIndex1d) ~ (index ||| rangeIndex1d).? ^^ {
          case matIdxY ~ matIdxX =>
          matIdxX.isDefined match {
                    case false  => Array[L4_Index](matIdxY)
                    case true   => Array[L4_Index](matIdxY, matIdxX.get)
          }
}
```

Listing 3.14: Parser extension to parser two brackets with ranges or indices

The `matIndex` can be added to an identifier in ExaSlang 4 programs, which is, together with other similar types of indices and access modifiers, added to the AST as a `L4_UnresolvedAccess`. The `matIndex` consists of a bracket with indices or a range, optionally followed by another bracket with indices and a range (line 1). Depending on the presence of the second bracket, the indices are added to an index-array (lines 2-6). The first bracket corresponds to indices in the y-direction, the second to the x-direction. The array is then inserted into the `L4_UnresolvedAccess` as a an optional attribute, which is set to `None` if it is not defined. For other accesses without brackets, the attribute is `None`.
The bracket access is implemented to access two types of variables:

- plain matrix variables
- fields of matrices

The resolve process for the two types looks very different because accesses to the first are simple and not modified much anymore after L4, whereas accesses to the second include many other access modifiers and processing steps. Both are at some point wrapped by an `IR/L4_MatAccess`-object, which does the mapping to the slice methods. For plain variable accesses, `L4_UnresolvedAccess`

are filtered on L4 and converted to another type of variable access, `L4_PlainVariableAccess`. One could carry through the bracket-operator by adding a new attribute to the access nodes, which is handed over between them until the bracket-operator is resolved in IR. But as most of them resemble a general access to variables, whereas the bracket operator corresponds to one certain type of accesses, which is accesses to matrix variables, this strategy should be avoided. Therefore, the following is done: on L4 for `L4_UnresolvedAccess`, whose `matIndex`-attribute is not `None`, are intercepted and converted to a new type of node, `L4_MatAccess`. A `L4_MatAccess` wraps plain variable access to conveniently store the information for a variable access, which is also necessary as a matrix variable access is a special type of variable access. The `L4_MatAccesses` are then progressed to IR, where they come together with `MatAccesses` from matrix fields.
The second type of accesses variables are fields of matrices. Accesses to these are complicated and their processing consists of many steps. Additionally, in contrast to plain variable accesses, they can contain many other access modifiers, which have to be processed. They are also filtered on L4 from `L4_UnresolvedAccess` and progressed to IR. To avoid interfering with the other processing steps of field accesses by wrapping it with another node or replacing it, as is done for plain matrix variables, having also to adapt the processing steps, the carry-through strategy is chosen here. The `matIndex`-attribute is added to field access nodes on L4 and IR. `matIndex` is handed over between different field access nodes in their processing pipeline. Not until in IR, also field accesses are wrapped with `IR_MatAccesses`.
After bracket-operators from different sources are all assembled as `IR_MatAccesses`, a strategy searches statements for them and converts them to the slice function nodes with the following policy:

- check if the `IR_MatAccesses` is at the left-hand side of an assignment, or at the right-hand side/an operand. Depending on this, mapping is done to `setSlice/setElement` or `getSlice/getElement`
- check the matIndex-array: if it contains two indices, map to `set/getElement`, for at least one range, map to `set/getSlice`
- replace the whole assignment for `setSlice/setElement` or only an expression for `getSlice/getElement`

The slice nodes are then processed the same way direct slice calls from L4 are.

3.4 Matrix Shape Classification

3.4.1 Motivation

The operations inversion and solving of linear systems highly depend on the structure or shape of the system matrix. These shapes each possess individual attributes: the information about a block-diagonal shape of a matrix can only be exploited if it comes together with information about the size of the diagonal blocks. Similarly, there are additional attributes connected to Schur-shapes, like the size of the A-block, n, and the size of the D-block, m. To transport information about any shape a matrix might have within the generation process, a generic class describing them is useful. The class should also be able to hold variable numbers of attributes with a different meaning, which makes it possible to define various different shapes and attributes as well as e.g. embedded shapes. Here, the A-block within the system matrix also has a specific shape with a specific attribute, namely block-diagonal with blocksize 3.
Moreover, a user may not know or predict the shape, a matrix to invert or a matrix arising from a `solve locally`-statement has, especially not with all the detailed information about e.g. block sizes. Automatic classification of the shape, a matrix has, solves this.
In this work, a classification, which can be used at compile-time to choose a suitable algorithm for compile-time or generate the algorithm for run-time to invert a matrix or solve a small linear system of equations is introduced. In a first approach to classifying, the matrix is inspected by iterating it at compile-time and checking for zero entries to identify zero-filled areas, as well as details of the shape, like block sizes. Hereby, the problem of variable accesses comes up, as described in section Symbolic Operation Example: to find zero blocks to exploit, zero entries have to be identified. If a matrix visited in the AST is present as variable access, this is not directly possible. A workaround for

that is to search for the declaration of the matrix variable and extract the initialization expression, composed of constant values. If no writes to the variable take place between its declaration and its classification, inversion or use as a system matrix, that is if the matrix stays constant, one can just operate on the initial expression. To enable this approach, a special collector is used.

3.4.2 Description of Shapes

From L4, matrix shapes can be specified by passing a list of key-value-pairs together with:

1. a field declaration of a matrix field
2. a matrix variable declaration
3. a `solveMatSys` statement
4. an inverse call

Each key names one feature of the matrix shape and the corresponding value defines the value, the feature is set to. In listing 3.15, an inverse call is told that the matrix to invert has block-diagonal shape with blocksize 3, and a matrix variable `t2` is specified as a diagonal matrix.

```
Var t_inverse : Matrix<Real, 6, 6> = inverse(
        t,
        "shape=blockdiagonal",
        "block=3"
)
Var t2 : Matrix<Double, 3, 3> = {
        {1.123, 0, 0},
        {0, 2.156, 0},
        {0, 0, 3.135}
} {shape=diagonal}
```

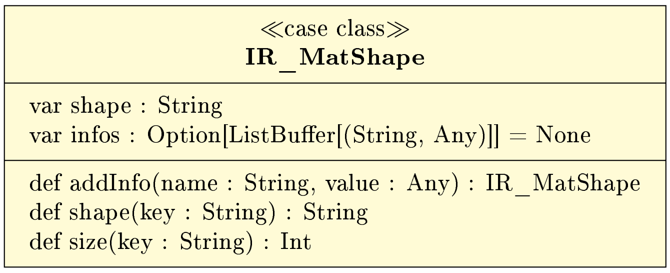
Listing 3.15: Different ways to pass a matrix shape on L4

The options 1-3 for passing a shape expect the shape specification in curly brackets after the construct declaration, while for inverse calls, it is passed as string function arguments. The attribute `shape` defines the highest order shape of the matrix, which is the macroscopic shape a matrix has and most of the time the only shape that defines the matrix. In listing 3.24, this is a schur-shape, as evident. If no shape is provided, the highest order shape is set to `filled`, which signals the absence of zero blocks. Furthermore, the classification of the matrix shape can be demanded by passing a key-value-pair: `detShape=compiletime` tells the compiler to determine the shape of the matrix at compile-time. A classification at run-time is also implemented in this work but left out because it makes a classification together with specialized inversion take as long as inverting the filled matrix. The key-value-pairs can be passed on L4 in any order. The corresponding part of the L4 parser is displayed in listing 3.16.

```
matShapeOption =
"{"      ~> repsep((ident <~ "=") ~ (ident | integerLit), ",") <~ "}"
         ^^ { case args => L4_MatShape(
                  args.to[ListBuffer].map(
                          s => IR_StringConstant(s._1 + "=" + s._2)
                  )
         )
}
```

Listing 3.16: Parser for matrix shapes

It defines a parser, `matShapeOption`, for repeated and comma-separated (`repsep` combinator) key-value-pairs composed of an identifier for the key followed by `=` and a value in form of another identifier or an integer literal. This is the meaning of line 1/2. After discarding curly brackets, each pair is mapped to one `IR_StringConstant` and a `L4_MatShape` is constructed from them in the rest of the listing. The `L4_MatShape` only holds the key-value-pairs and progresses them to IR. Internally, the attributes are handled within a list. When progressing to IR, the pairs are added to the list as a (`String`, `Any`)-pair, which involves separating the key-value-pair from the `IR_StringConstant` again with a *regex*. A regex is a string pattern, which is used to extract parts of a string to variables. `Any` is the supertype of all types in scala. It is used to make it possible to associate any value with a key. Also, attributes can be added later on in the compilation process. At all times, the operation of adding a pair to the list is executed by the method `addInfo`. `addInfo` returns the `IR_MatShape`, the value has been added to, to enable chaining. When an attribute is demanded from the `IR_MatShape`-object, the list is searched for the corresponding value, which is then returned. For returning attribute values, there are two modes: for values of block sizes or other attributes that consist of an integer value one mode is used (method `size`), for attributes which values are string, another one (method `shape`) is used. The `shape`-attribute is always present and stored separately from the other attributes. Figure 3.1 depicts the class in detail. It is derived from `IR_Expression`. Defining the attribute list (`infos`) as a Scala `Option` makes it possible to



```
≪case class≫
IR_MatShape

var shape : String
var infos : Option[ListBuffer[(String, Any)]] = None

def addInfo(name : String, value : Any) : IR_MatShape
def shape(key : String) : String
def size(key : String) : Int
```

Figure 3.1: Class diagram of `IR_MatShape`

define a value or leave the attribute at `None`. So if no attributes exist, the list can be set to `None` to not carry around a list object without purpose. Using a hashmap instead of a list was considered but rejected because a plain list seemed more suitable for the low number of key-value-pairs.
Inside the internal inversion method, retrieving and using the attributes of an `IR_MatShape` looks as follows:

```
matShape.shape match {
case "diagonal"
        => {  /* ... calculation */  }
case "blockdiagonal"
        => {
                val blocksize : Int = matShape.size("block")
                // ... calculation
        }
case "schur"
        => {
                val n : Int = msi.size("block")
                val shapeA : String = msi.shape("A")
                val bsizeA : Int = msi.size("Ablock")
                // ... calculation
}
```

Listing 3.17: inversion based on the `matShape`-object

First, the macroscopic shape of the matrix is matched and depending on its value, an inversion algorithm is chosen, e.g. diagonal or blockdiagonal. In case the matrix has blockdiagonal shape, the value of the diagonal blocks' size is retrieved from the matrix shape object in line 6. The internal method which solves local systems directly looks similar in structure.

3.4.3 Classification Algorithm

In this section, the algorithms for automatically classifying different shapes for a given matrix are described. For each shape, the matrix is iterated completely and checked for a distinctive shape. This is implemented for diagonal, blockdiagonal and Schur shape, but can be extended with any other shape as well, if an algorithm to recognize it exists. For inversion and solving of linear systems the classification takes place right before execution of respective operation. Algorithm 3.2 describes the whole procedure. The options for `shapes` are diagonal, blockdiagonal and Schur. `isS()` is

Algorithm 3.2 Pseudo code for the matrix shape classification

```
function isOfShape(M) : matShape
    for all shape S in shapes do
        if (matShape ← isS($M$)) != matShape(shape=Filled) then
            matShape
        end if
    end for
    return matShape(Filled)
end function
```

implemented differently for each shape. Each individual shape classification returns a matShape object. If the contained macro shape in the returned object is not `Filled`, the shape `S` is recognized successfully and returned. This takes place in line 3-5. If none of the shapes is recognized successfully, a matShape object with `Filled` is returned.
For diagonal, the classification is depicted in algorithm 3.3. Diagonal shape classification is straight

Algorithm 3.3 Pseudo code for diagonal shape classification

```
function isDiagonal(M) : matShape
    for $i \leftarrow 0, N$ do
        for $j \leftarrow 0, N$ do
            if $i$ != $j$ and $M(i, j)$ != 0 then
                return matShape(shape=Filled)
            end if
        end for
    end for
    return matShape(shape=Diagonal)
end function
```

forward: if there is a nonzero off the diagonal, the matrix is not a diagonal matrix, so the default matShape `Filled` is returned. When no zero entries were found outside the diagonal, `Diagonal` is returned.
For blockdiagonal matrices, algorithm 3.4 is employed. The algorithm works the same way algo-

Algorithm 3.4 Pseudo code for blockdiagonal shape classification

```
function isBlockdiagonal(M) : matShape
    bsize ← 0
    while not all new entries in the block are zero do
        check new entries in block $M(0 : bsize, 0 : bsize)$ for zeros
        bsize ← bsize +1
    end while
    for $i \leftarrow 0, N$ do
        eoBlock ← ($i$/bsize)*bsize+bsize
        for $j$ ← eoBlock, $N$ do
            if $M(i, j)! = 0$ or $M(j, i)! = 0$ then
                return matShape(shape=Filled)
            end if
        end for
    end for
    return matShape(shape=Blockdiagonal,block=bsize)
end function
```

rithm 3.3 does: it checks for nonzero entries outside the pattern, that is the main diagonal or main

diagonal blocks with certain blocksize. Before that, a blocksize has to be found. In lines 2-5, the blocksize (`bsize`) is determined on the first potential diagonal block, which is done by incrementing the counter variable `bsize` and then checking for zeros in the added entries of the block defined by the counter-variable (lines 3-6). If only zero entries occured, the block ended and the block size is determined. After a blocksize has been found, all entries of the matrix that lie outside of the diagonal blocks are checked for nonzeros by calculating the end of each diagonal block (`eoBlock`) and iterating from there in x and y direction to the end of the matrix, searching for nonzero entries (lines 7-14).
For the Schur-shape, the blocksize of the block matrix D (`bsizeD`) is determined in the same way the diagonal blocksize is, but starts counting from the upper right corner of the matrix. If the upper left corner is reached, the matrix is filled and a corresponding matShape-object is returned. Furthermore, the other shape classifications are executed on the resulting block matrix A to classify its shape. In algorithm 3.5 the classification for Schur-shapes is displayed.

Beginning from the lower and right corner of the matrix, the counter variable `bsizeD` is incre-

Algorithm 3.5 Pseudo code for blockdiagonal shape classification

```
1: function isSchur(M) : matShape
2:     bsizeD ← 0
3:     while non all new entries in potential $B$ and $C$-blocks are zero do
4:         check new entries in blocks $M(0 : N - bsizeD, N - bsizeD : N)$
5:         and $M(N - bsizeD : N, 0 : N - bsizeD)$ for zeros
6:         bsizeD ←bsizeD+1
7:     end while
8:     if $N-$ bsizeD == 0 then
9:         return matShape(shape=Filled)
10:    end if
11:    shapeA ← isOfShape($M(0 : N-\text{bsizeD}, 0 : N-\text{bsizeD})$)
12:    return matShape(shape=Schur, block=N - bsizeD, shapeA)
13: end function
```

mented and new entries to the B and C-blocks are checked (lines 3-7). If the upper border of the matrix is not reached (lines 8-10), also the resulting A-block is classified (line 11). The classification of A is added as inner shape to the returned matShape-object (line 12).
Figure 3.2 visualizes the classification algorithms. A Schur-shape is recognized in a `N` × `N` matrix. `bsizeD` denotes the blocksize of the D-block. Additionally, a blockdiagonal shape is recognized in the block matrix A, delimited by `bsize` × `bsize`, which again has diagonal blocks of size `bsizeA` × `bsizeA`. Arrows symbolize loops.

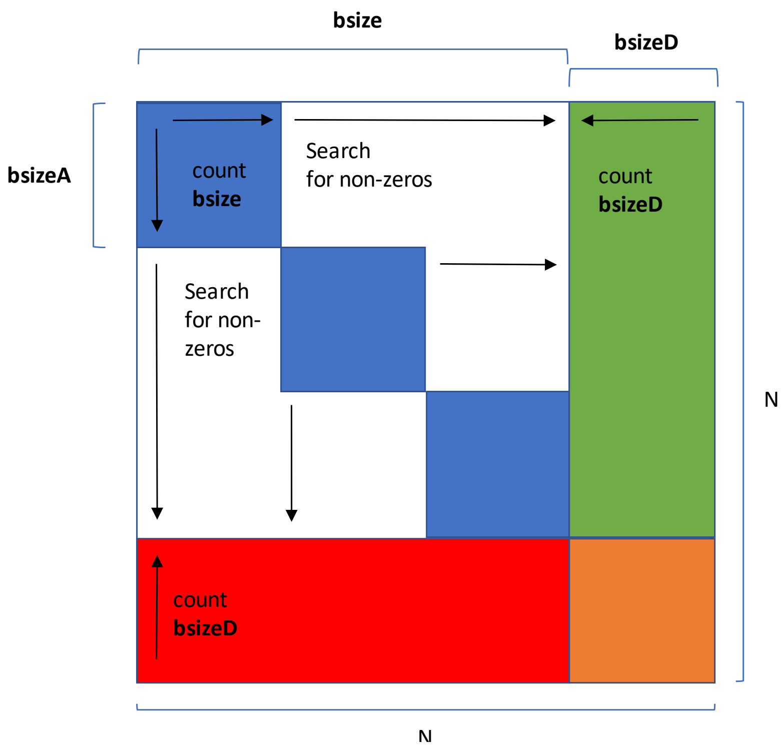


Figure 3.2: Schematic image of a schur-shape classification

3.5 Specialized Inversions

3.5.1 Motivation

The inversion is part of the version previous to this work. Inversion of an L4-matrix can be called in the same manner as the transpose-call of example 2.8. Inverse-calls are parsed as function-calls to L4 and progressed to the IR. There, a strategy searches and replaces them with the result of an internal method that carries out the calculation. Furthermore, this internal inverse-method is used to solve the local systems of `solve locally`-statements in the previous version of the framework by inverting the system matrix. As this is, in some cases, not an effective way to solve a linear system, it was changed as described in section Solving Local Linear Systems. However, if a local linear system is solved many times within a loop, whereby the system matrix stays the same and only the right-hand-side changes, it is useful to calculate the inverted system matrix once and draw it in front of the loop. Instead of solving a linear system per iteration, only a matrix-vector-multiplication has to be executed additionally to a single inversion of the system matrix.
The inversions stemming from `solve locally`-statements, as well as direct inversion-calls on L4, can be determined to run at compile-time or run-time by specifying an attribute in the knowledge file. Knowledge files contain flags and attributes to specify macroscopic parameters of the code generator.
Specialized inversion labels the exploitation of zero blocks in the matrix during its inversion, leading to less calculation. For instance, inversion of a $n \times n$ block-diagonal matrix, only small diagonal blocks have to be inverted instead of the full $n \times n$ matrix.
There are three advantages of using a specialized inversion algorithm:

1. reduced compile-time due to less calculation for compile-time inversion
2. reduced run-time due to less calculation for run-time inversion
3. reduced node numbers for compile-time inversion

The required time for inversions depends on how many inversions the program contains, but for

an example application the compile-time inversion can make up to 20 percent of the compile-time. As the `solve locally`-calculations are a reasonably computation-expensive part of the generation process as well as of the run-time of the application in case of run-time execution, investing the effort to implement specialized algorithms is justified. The achieved speed up by specialized inversion is presented in section Results for both run-time and compile-time execution.
The third advantage is the number of nodes that are added to the AST during a compile-time inversion. For every arithmetic operation used within the inversion, a new node is produced, e.g. $a+b$ becomes `IR_Addition(a,b)`. Additionally, multiple accesses to one entry in the input matrix are each represented as an `IR_VariableAccess`. Each index thus produces a Scala-object with attributes and administrative structures. Therefore, the more calculations, the more objects have to be built and saved in the AST and the larger the AST becomes. Therefore, having to calculate less is worth not only because of the achieved speed up. More calculations lead to (i) at least prolonged generation time due to copying of large expression within the calculation and in the worst case to a stack overflow, (ii) longer traversal times for all strategies, as the AST is larger and resulting from that also prolonged generation time and (iii) longer expressions to be evaluated at run-time of the application. To demonstrate this, figure 3.3 depicts the sub-tree for one entry of a compile-time inverted 2×2 matrix within the AST, according to listing 2.12. The generated C++-code reads $(1.0/((m[0]*m[3])-(m[1]*m[2])))*m[3]$, involving a linearization of the access indices. This gives a preview of the AST size of an inverted 5x5 matrix: for larger matrices, the entry expressions are larger and there are more entries, thus the AST explodes in size.

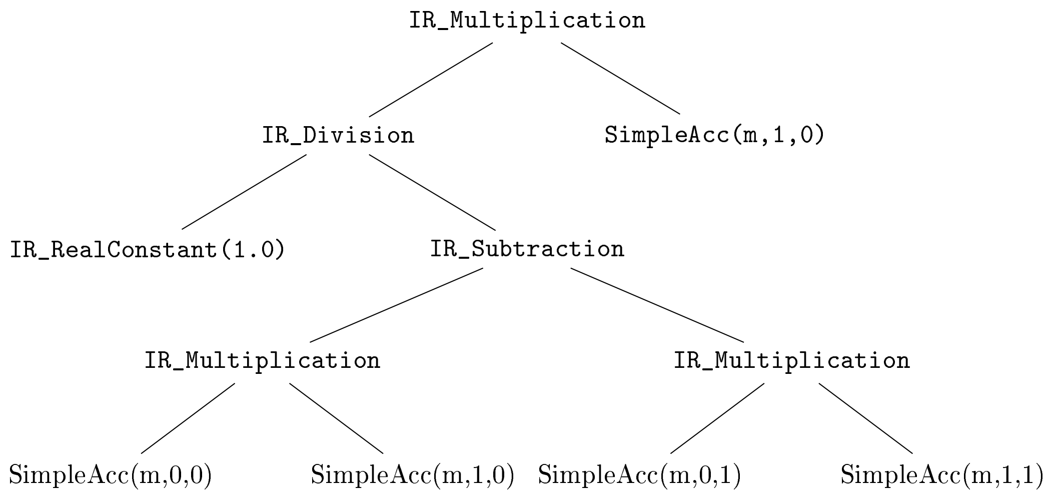


Figure 3.3: AST structure the expression of one entry in an inverted matrix, `IR_VariableAccess` are simplified to SimpleAcc(variable name, row index, column index)

3.5.2 General Inversion

The *general inversion* is the inversion of matrices, which are not of any special shape defined by zero-blocks, but completely consist of nonzero elements. It is realized by LU-decomposition for run-time inversion and by cofactor inversion or LU-decomposition at compile-time. The schemes of sections Blockdiagonal Inversion and Schur Inversion to invert a matrix both include the inversion of smaller sub-matrices. They are also inverted by LU-decomposition. Matrices without zero blocks are called *filled*.
In the previous state of the generator, general inversion at compile-time is executed either by the Gauss-Jordan algorithm or by calculating matrices of cofactors. The first method is a form of gaussian-elimination, as LU-decomposition is. In this work, LU-decomposition is preferred, because it can also be used to solve linear systems without an inverse. Both applications are based on LU-decomposition, so they use the same internal function for calculating an LU-decomposition, and then branch to calculating their respective targets, an inverted matrix or a solution vector. The second method was first replaced due to its high computational complexity. Later on, this way of inverting a matrix showed advantages over LU-decomposition if executed at compile-time, which

is why it is set as a standard approach for some cases again. These advantages are discussed in section Results.
Nevertheless, LU-decomposition is always used for run-time inversion and compile-time inversion in some cases. This is the same in the previous state of the generator, though the pivoting is not working. As the LU-decomposition is also used to solve small linear systems directly at compile-time and run-time, which is implemented from scratch in this work and described in section Solving Local Linear Systems, the algorithm is newly implemented to suit these applications and fix the pivoting. Algorithm 3.6 implements an LU-decomposition followed by the forward and backward substitution of listings 2.3 and 2.4.

Algorithm 3.6 Inversion by LU decomposition and forward backward substitution

```
1:  function LUPINV(A,AInv)
2:      init P
3:      LUPDECOMP(A,P)
4:      for $j \leftarrow 0, N$ do
5:          for $i \leftarrow 0, N$ do
6:              if $P(i) == j$ then
7:                  $AInv(i)(j) \leftarrow 1$
8:              else
9:                  $AInv(i)(j) \leftarrow 0$
10:             end if
11:             for $k \leftarrow 0, i$ do
12:                 $AInv(i)(j) \leftarrow AInv(i)(j) - A(i)(k) * AInv(k)(j)$
13:             end for
14:         end for
15:         for $i \leftarrow N-1, 0$ do
16:             for $k \leftarrow i+1, N$ do
17:                 $AInv(i)(j) \leftarrow AInv(i)(j) - A(i)(k) * AInv(k)(j)$
18:             end for
19:             $AInv(i)(j) \leftarrow AInv(i)(j)/A(i)(i)$
20:         end for
21:     end for
22: end function
```

Firstly, a permutation array is initialized with references to all rows, followed by the LU-decomposition of algorithm 2.2. P saves row permutations. If for later calculations the right hand side of a linear system is included, it is accessed through P. This maps the access to the correct row, taking row permutation into account. Then, the following is done for every column of the system $AX = I$ (j-loop), implementing definition 2.6: a forward substitution is employed in lines 5-14, while also initializing the current column of the permutated unit matrix of the system. Afterwards a backward substitution is executed for the current column (lines 15-20). That way the inverse of A, i.e. X, is obtained column by column, which takes $\frac{2}{3} * n^3 + n * 2 * n^2 \approx \frac{8}{3} * n^3$ flops for a quadratic matrix of size $n \times n$.
At compile-time, matrices are present as `IR_MatrixExpression`. The algorithm is executed by setting up new arithmetic operator nodes according to the pseudo code, in the manner illustrated in section Symbolic/Compile-time inversion. One can consider the execution of the loops in algorithm 3.6 at compile-time as an unrolling of the assignments of the whole iterations space in the destination variable. Afterwards, e.g. in entry $AInv(0)(0)$, all assignments in the algorithm to $AInv(0)(0)$ are present as arithmetic operation nodes.
Generating the code for a run-time execution of algorithm 3.6 consists of setting up a series of `for`-loop, `if`-statement, arithmetic operator and access nodes. The node collection is printed, compiled and executed at run-time of the application. A short excerpt from the code to generate algorithm 3.6 is given in listing 3.18.

```
IR_ForLoop(
        IR_VariableDeclaration(k, i + 1),
        IR_Lower(k, N),
          IR_PreIncrement(k),
          ListBuffer[IR_Statement](
                    IR_Assignment(
                              SimpleAcc(out, i, j),
                              IR_Subtraction(
                              SimpleAcc(out,i,j),
                              IR_Multiplication(SimpleAcc(in, i, k),
                              SimpleAcc(out, k, j))
                    )
          )
)
```

Listing 3.18: abstract from the LU generator code

It implements line 16-18 of algorithm 3.6 by building a `for`-loop node from nodes for initialization, boundary condition and increment of the loop index variable. Next, the loop is body is extended by the assignment of line 17. `IR_HighDimAccess` is simplified to `SimpleAcc` to make the code more understandable. `out` is a variable access to the target variable, that holds the result of the calculation, `in` an access to the input matrix. During the generation process, an `IR_FunctionCall` for an inverse is replaced by, among further statements, the nodes of this `for`-loop.

Inverting Blocks of a Matrix

For a block matrix, there are two options to invert a single matrix block only. The first option is *out of place*, by copying the block to a separate matrix, inverting it and writing it back to the block matrix at the correct location. This is done for blockdiagonal and Schur inversion at compile-time. The second option is to employ the inversion locally on the block matrix, i.e. *in place*. Here, the location of the block and its size are also passed to the internal inversion-method. This avoids allocating memory locally or implementing a memory allocation at the beginning of the application for all matrix blocks. In place inversion is implemented for run-time blockdiagonal inversion. The diagonal blocks are inverted by the method `localLUPINV`, which receives the position of the block to invert as `offset_r` and `offset_c`. Its size is passed as `blocksize`:

```
def localLUPINV(
          in : IR_Access,
          blocksize : Int,
          offset_r : IR_Expression,
          offset_c : IR_Expression,
          out : IR_Access
) : ListBuffer[IR_Statement] = { ... }
```

Listing 3.19: signature of a LU-inversion-call

All accesses to `in` and `out`, as well as loops are then shifted by the offsets.

Problems when Pivoting at Compile-time

In the generated C++ code for the LU-decomposition, pivoting is implemented without further complications. An LU-decomposition at compile-time however, or more specific, the pivoting that is involved, needs special treatment. The decomposition is executed symbolically, hence exact values of entries are not necessarily known. That makes the maximum-search of line 3 in algorithm 2.2 more complicated, because the comparison is not always possible, depending on the form of the entries. Within the AST, entries can be of three different forms:

1. accesses
2. constant values
3. calculations (e.g. addition of two constant values or accesses)

If an access is encountered, its exact value can not be extracted, so accesses can not be compared to accesses. Constant values within the AST can be compared as values in a normal program can. Calculations can be evaluated in the manner of compile-time calculations as described in section Run-time and Compile-time Execution if they completely consist of constant values. Accordingly, they can also be compared. Otherwise, they can not be evaluated and compared in the maximum-search.
This is resolved as follows:

- If there are only constant value entries and computations of constant values, the maximum-search is executed as usual. For a very small maximum, the matrix degenerates. Accesses and computations of accesses are skipped.
- Furthermore, if the maximum-search finds a very small element and there were accesses skipped or if there are only accesses and computations of accesses, the first access found is chosen as the pivot element.

The chosen access might be a very small value or zero when evaluated at run-time. To prevent numerical instability or an exception due to division with zero, the chosen pivot access can be checked at run-time. Activated with a knowledge flag, this check involves copying and saving all pivots at compile-time and generating them as an array. At run-time, the pivot accesses' exact values can be evaluated and an error is produced if they are very small. In this way, access entries can be used at least for some cases.

Pivot-Elimination-Optimization for LU-Decomposition

To reduce the expression length of LU-decomposition-based operations in the case of matrices filled with accesses, an optimization can be applied: saving the pivot-elements to separate variables, similarily to the check making sure pivot elements are not very small, described in the previous section. For this optimization, the pivot expressions are not only duplicated and saved but also replaced by accesses in the matrix to invert. Due to the processing of LU-decomposition, in particular, multiplying rows by one pivot-element, the expressions of the pivots occur multiple times in the inverted matrix or solution of the linear system. Therefore, eliminating them by replacement with variable accesses leads to a reduction of nodes in the AST.

3.5.3 Blockdiagonal Inversion

Blockdiagonal matrices are inverted by inverting the main diagonal blocks. In pseudocode, this looks as follows: Every block is inverted and written to the result matrix at the corresponding

Algorithm 3.7 Pseudo code for blockdiagonal matrix inversion

```
1: function BLOCKDIAGONAL(A)
2:     for block in diagonal blocks of A do
3:         INV(block,AInv)
4:     end for
5: end function
```

position. This is straight forwardly implemented for execution at compile-time and generation for run-time.

3.5.4 Schur Inversion

The inversion by Schur-complement of a matrix M following definition 2.1.1 includes the inversion of the matrix block A. Afterwards, the Schur-complement S is calculated, followed by calculation of the right hand side of definition 2.1.1. In matrices arising from `solve locally`-statements, the matrix block A often has blockdiagonal form. This can again be exploited as described in the previous section. Algorithm 3.8 implements this approach. First, the matrix blocks are extracted from their positions in M, indicated by 2D-intervals, e.g. the upper left block A by `0:n,0:n`. Following this, A is inverted depending on its shape. At last, the Schur-complement and all other terms that are involved in the inversion are calculated and written to the result matrix $MInv$.

Algorithm 3.8 Algorithmic formulation of the inversion by schur-complement.

```
function SCHUR(M,MInv)
    $A \leftarrow M(0:n, 0:n)$
    $B \leftarrow M(0:n, n:n+m)$
    $C \leftarrow M(n:n+m, 0:n)$
    $D \leftarrow M(n:n+m, n:n+m)$
    if shape(A) == blockdiagonal then
        $AInv \leftarrow$ BLOCKDIAGONAL(A)
    else
        $AInv \leftarrow$ INV(A)
    end if
    $S \leftarrow D - C * AInv * B$
    $SInv \leftarrow$ INV(S)
    $MInv(0:n, 0:n) \leftarrow AInv + AInv * B * SInv * C * AInv$
    $MInv(0:n, n:n+m) \leftarrow -AInv * B * SInv$
    $MInv(n:n+m, 0:n) \leftarrow -S * Inv * C * AInv$
    $MInv(n:n+m, n:n+m) \leftarrow SInv$
end function
```

Like algorithm 3.7, the Schur inversion is straight forward implemented for compile-time. For run-time, the decision about how to invert the A-block is made at compile-time.

SchurWithHelpers-optimization

Even though applying a Schur-complement to invert a matrix at compile-time reduces the complexity, expressions grow large quickly due to the load of calculations stemming from the matrix-matrix-multiplications in definition 2.1.1. This can be further reduced by generating the common sub-expressions in definition 2.1.1, such as S^{-1} or $A^{-1}BS^{-1}$ as separate declarations of *helper-variables* and letting the following calculations at compile-time access the helpers instead of using the long expressions of the helpers entries and copying them if they are used for multiple other calculations. This implements the same idea as the common sub-expression-elimination given in listing 2.1: common sub-expressions of the different entries in definition 2.1.1 are saved to variables and the subsequent calculation of the entries uses only accesses to the helper-variables. The optimization is activated by a knowledge-attribute, and implemented as follows:

- Within the Schur-inversion for compile-time the helper-variables like S^{-1} are copied and saved in an `IR_VariableDeclaration` after their calculation.
- The helper declarations are furthermore annotated to the inverted matrix expression.
- The calculationusing the values of the helper-variables proceeds with accesses to the helper-variables.
- A strategy finds statements containing matrix-expressions with annotated helpers and puts their declarations in front of the statement. At run-time, the expressions of the inverted matrix access the declarations.

The impact of this procedure on the AST-size is demonstrated in section Results. A snippet of the Schur-inversion of a 3×3 matrix, whose A-block is a 2×2 diagonal matrix is given in listing 3.20.

```
__matrix_double_3_3_t schur_mat { /* initialization */ };
__matrix_double_2_2_t A_inv_0 {
          (1.0/schur_mat[0]),
          schur_mat[1],
          schur_mat[3],
          (1.0/schur_mat[4])
};
__matrix_double_1_2_t CA_inv_0 {
        ((schur_mat[6]*A_inv_0[0])+(schur_mat[7]*A_inv_0[2])),
        // ...
};
__matrix_double_2_1_t A_invB_0 {
        ((A_inv_0[0]*schur_mat[2])+(A_inv_0[1]*schur_mat[5])),
        // ...
};
/* ... other subexpressions */
__matrix_double_3_3_t schur_mat_inv {
        /* ... calculation of the inverse from subexpressions */
};
```

Listing 3.20: Compile-time inverted schur matrix and its helper-variables

In lines 2 - 7, one of the expressions occuring multiple times in the Schur-inversion, the inverted A-block, is calculated by inverting the diagonal entries of the input matrix `schur_mat`. Following expressions of the Schur-inversion, e.g. CA^{-1} (lines 8-11) and $A^{-1}B$ (lines 12-15) are calculated from the A^{-1}-variable, instead of using the same copied expressions of the inverted A-block shown in line 3 - 6 multiple times. The zero at the end of each variables name is used to distinguish between helper-variables of different Schur-inversions.

3.5.5 Initialization Expressions

Entry $M(i, j)$ can not be known at compile-time, if is present as a variable access or `IR_HighDimAccess`. Consequently, in this case pivoting can not take place effectively. Also the matrix classification of section Matrix Shape Classification relies on evaluating entry values and therefore has problems with non-constant variable accesses. But with a special collector, which can retrieve the initialization expression of the matrix variable to be classified, as well as provide insurance that there were no writes to the variable, this is solved if the initial expression exists. The initialzation expression is often available in the AST as a collection of constant numbers, which can be evaluated. Accesses can also be given in the initial expression already if the matrix is initialized with another variable. Then, the entries that are constant can be used. There must not be a writing access to the variable between the declaration and classification or compile-time LU-decomposition, so the expression can be assumed as constant. To achieve this, a variable declaration collector like in the example from section Collectors is extended to also store write accesses, by extending its enter-method as follows:

```
override def enter(node : Node) : Unit = {
node match {
        // add declaration to decl-collection
        case d : IR_VariableDeclaration                 => addDecl(d)
        // open new scope
        case _ : IR_ForLoop                             => openNewScope()
        case _ : IR_Scope                               => openNewScope()
        // ... other scope opening constructs follow
        // Extension: add write to write-collection
        case _ @ IR_Assignment(dest, _, _)              => addWrite(dest)
        // ... other ways to write a variable follow
        case _                                          =>
}
}
```

Listing 3.21: enter method of a collector

The encountered node `node` is matched for its type. In line 2-8 the default functionality of a variable declaration collector is implemented. From line 9 on, language constructs that can change their inputs, e.g. an assignment to a variable `dest` are matched and the variable written to is saved in a write-access-collection. This collection is organized in the same way as the variable-declaration-collection is. A constant initial expression can now be retrieved from the collector by handing it the variable's name. The collector then checks if a declaration exists for this name and furthermore checks if a write-access to that variable took place. If that is not the case, the initial expression, if available, is returned. If a write was found, `None` is returned.

3.6 Solving Local Linear Systems

3.6.1 Motivation

To solve small linear systems of equations of the form $Ax = b$ directly without calculating the inverse, a new type of node is introduced. Its task is to take a system matrix A, right-hand side b and unknowns x as accesses, solve the system, and write the solution to the unknowns according to definition 2.5.
If only a single system has to be solved, solving the systems without calculating an inverse is more effective, which can be shown by the following comparison of approaches: (i.) solving a system by calculating the inverse of the system matrix with LU-decomposition and multiplying it with the right-hand side and (ii.) solving a system by calculating an LU-decomposition and employing a forward and backward substitution. (i) takes $\frac{8}{3}n^3 + n^2 \approx \frac{8}{3}n^3$ flops, whereas the first term stems from algorithm 3.6 and the second term from the matrix-vector multiplication. (ii) takes only $\frac{2}{3}n^3 + 2 * n^2 \approx \frac{2}{3}n^3$ flops. Here, the source of the first term is the LU-decomposition and the second terms source are the substitutions.
Therefore, systems that are passed to this node are by default solved by calculating an LU-decomposition of the system matrix, followed by the application of forward and backward substitution on the resulting triangular systems. The same way inversion can be optimized by exploiting zero-blocks in the matrix, algorithms to solve linear systems of equations can be made more effective, which is implemented in this work for block-diagonal system matrices and Schur system matrices with certain restrictions.

3.6.2 SolveMatrixSystem

The functionality of solving small systems directly can be used in two ways within the generator:

1. Via a new statement on layer 4, to which the system matrix, as well as unknowns and right-hand-side, are passed as variable accesses.
2. Systems constructed within `solve locally`-statements can be solved if the statement is not enclosed by loops. This takes place in IR.

To make the first option possible, the L4 parser is extended by:

```
solveLinearSystemStatement =
        ("solveMatSys") ~>
        (expression <~ ",") ~
        (expression <~ ",") ~
        expression ~
        matShapeOption.?
{
        case A ~ x ~ b ~ shape => L4_SolveMatrixSystem(A, x, b, shape)
}
```

Listing 3.22: Parser code for `SolveMatSys`-statement

In this parser, two new combinators occur: ~> and <~. They both parse their two operands and discard the operand they do not point to. .? marks an optional parse: A single operand may be parsed or may not. Thus the complete absence of a input matching the pattern of the parser also counts as an successful parse. `expression` parses expressions of various types, in particular variable accesses. `matShapeOption` is a parser for matrix shapes. It is discussed in section Matrix Shape Classification. With these parts, the new parser parses: a new keyword, `solveMatSys` (line 2), followed by three comma-separated expressions, which are the accesses for A, x and b (lines 3-5) and optionally the language construct to define a matrix shape (line 6). It discards all delimiters and the new keyword in the process. After matching the results, a `L4_SolveMatrixSystem` is

constructed (line 8).
The new node holds the accesses and shape on L4. After progressing to IR, a strategy finds the corresponding `IR_SolveMatrixSystem` and executes the embedded function which solves the system depending on the shape of the system matrix. If it has no particular shape, a LU-decomposition in form of algorithm 2.2 is executed, followed by the forward and backward substitution formulated in algorithms 2.3 and 2.4. A new attribute in knowledge-files determines, if the system is to be solved at run-time or at compile-time. For the first option, the algorithms are generated as C++-code, for the second, they are executed in the AST at compile-time.
On L4, the new statement looks as follows:

```
Var A : Matrix<Real, 3, 3> = {{3,2,-1},{2,-2,4},{-1,0.5,-1}}
Var x : Matrix<Real, 3, 1>
Var b : Matrix<Real, 3, 1> = {{1},{-2},{0}}
solveMatSys A,x,b
```

Listing 3.23: the new statement in ExaSlang 4

To be able to pivot in the LU-decomposition, the generator tries to find the constant, initial expression for the system matrix A, to pivot a matrix with mostly constant entries, which can be evaluated. After the `solveMatSys`-statement in line 4, the solution of the system `Ax = b` is present in the unknowns variable `x`.
The second way of using the node takes place inside `solve locally`-statements. A local system matrix and right-hand-side are built of values from the equations passed with the `solve locally`-statement. They are then given to an `IR_SolveMatrixSystem`-node, from where on the node is resolved as described before. Additionally, statements to write the local solution back to the solution field are generated, instead of a single assignment writing the solution to the unknowns variable `x`.

3.6.3 Special Systems

Blockdiagonal systems

For a blockdiagonal system matrix A, the system can be decomposed to several smaller component systems, which are then solved. Suppose the following system $Ax = b$:

$$\begin{pmatrix} A_{11} & 0 & 0 & 0 \\ 0 & A_{22} & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & A_n \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \tag{3.1}$$

where A_{ii} are sub matrices and b_i and x_i are vectors. For every unknown block x_i, a component system

$$A_{ii}x_i = b_i \tag{3.2}$$

is set up and solved by LU-decomposition and foward-backward-substitution. Afterwards, the solution vector x is composed of the calculated solutions x_i from the component systems. At compile-time, a component system is set up for every diagonal block and the corresponding unknowns and right-hand side by extracting them to their own matrix expression and applying the solution algorithm at compile-time. At run-time, the smaller systems are set up as separate variables. Then, a generated version of the solution algorithm is used.

Schur systems

In case the system matrix has a suitable shape to apply the Schur-complement, the matrix decomposition described in section Fields and `solve locally` is employed. It is extended to arbitrary block numbers and sizes in this work, although it is still restricted to $m = 1$. Hereby, m is the width of the D-block in the Schur-representation. In this way, the system matrix is considered as a block matrix composed of the blocks A, B, C, and D, which are used to form the Schur complement matrix S and solve the system defined by S for the unknowns.

The shape of the system matrix can be communicated to the generator on L4. In listing 3.24, a system with a 7×7 matrix is solved. The system matrix has Schur-shape.

```
1  Var M : Matrix<Real , 7, 7> = {
2          {1 , 2, 1, 0, 0, 0, 3},
3          {4 , 2, 1, 0, 0, 0, 1},
4          {1 , 5, 1, 0, 0, 0, 5},
5          {0 , 0, 0, 4, 1, 3, 2},
6          {0 , 0, 0, 1, 1, 2, 3},
7          {0 , 0, 0, 4, 6, 7, 3},
8          {3 , 5, 7, 4, 4, 5, 1}}
9  Var x : Matrix<Real , 7, 1>
10 Var b : Matrix<Real , 7, 1> = {{1},{-2},{0},{2},{1},{4},{1}}
11 solveMatSys M,x,b {shape=schur,block=6,A=blockdiagonal,Ablock=3}
```

Listing 3.24: SolveMatSys for a Schur system matrix

In line 11, `SolveMatSys` is told, that the shape of the given matrix is a Schur-shape, as well as that the A-block is of size 6×6 and has block-diagonal shape with 3×3 blocks. This describes the shape of `M` in the snippet.

3.7 User Guide

A summary of how to activate the implemented features is given in the following. The generator can be advised to use a certain algorithm or exploit a present matrix shape in the matrix operations in two ways:

- Knowledge attributes
- in the ExaSlang 4 program

Knowledge attributes are specified in a knowledge-file, which is given to the generator and contains additional information and parameter values necessary for a successful program generation. They are defined like normal variables, except they do not possess a type in the knowledge file. An example is to demand a run-time or compile-time execution of an inversion or the solution of a small linear system:

```
experimental_resolveInverseFunctionCall = "Compiletime"
experimental_resolveLocalMatSys = "Compiletime"
```

Listing 3.25: two knowledge attributes

There is one other option for both of them: "Runtime". All attributes listed here are set and unset in the same way:

- In case of compile-time inversion of a Schur-matrix, the SchurWithHelper-optimization can be activated via a knowledge attribute: `experimental_SchurWithHelper = true`, as well as the Pivot-Elimination-optimization when decomposing a matrix at compile-time per LU-decomposition: `experimental_CTPivotElimination = true`.
- For the same case, checking pivot elements used in a compile-time LU-decomposition at run-time is switched on by: `experimental_checkCTPivots = true`, as well as the used tolerance in the check: `experimental_CTPivotTolerance = doubleValue`.
- `experimental_classifyLocMat = true` causes a classifcation of system matrices of `solve locally`,- or `solveMatSys`-statements, if a constant expression for it can be retrieved.
- The shape of a system matrix in a `solve locally`-statement can be, if known in advance by the user, provided by the knowledge attributes `experimental_locMatShape`, `experimental_locMatBlocksize`, `experimental_locMatShapeA` and `experimental_locMatBlocksizeA`. This is only a provisionally solution and a mechanism to pass shapes of `solve locally`-statements similar to passing matrix shapes of `solveMatSys`-statements might be added in the future. It is also restricted to Schur,- and blockdiagonal

shapes. However, shapes can be provided to the `solve locally`-mechanism also by defining it in the field declaration of the iterated field.

Inversion-calls and `solveMatSys`-statements offer an interface pass a matrix shape or the demand to classify the matrix. This is achieved by a comma separated list of key-value-pairs:

```
Var inv : Matrix<Real, 6, 6> = inverse(m, "shape=blockdiagonal", "block=3")
solveMatSys A,u,f {shape=LU}
```

Listing 3.26: Examples for parameters defined for inversion and linear system solution

As of now, only the here mentioned keys carry meaning, but in the future more keys can be implemented easily to cause the execution of a specific algorithm in the generator. This is part of the design idea of `matShape`-objects. `shape` is the key that always has to be defined when defining a matrix shape. It can be set to: `filled`, `diagonal`, `blockdiagonal` and `schur`. For `blockdiagonal`, also a `bsize` has to be set. For `schur`, `bsize` and `A` have to also be provided. The classification of a matrix shape is activated by setting the key `detShape` to `compiletime`. In inversion calls, specifying specific algorithms for the inverion is also possible for the key `shape`: `cofactors`, `gaussJordan`, `LU` issue the named algorithms. The default algorithm used is cofactor for matrices of accesses and LU for constant matrices. After a `solveMatSys`-statement, `LU` and `QR` enable the use of LU,- and QR-decomposition to solve a linear system. Note that only LU is implemented and testet for run-time and compile-time, and QR only implemented for compile-time. The QR-decomposition should not be used yet, as its implemented for experimental purposes, see section Conclusion for Matrices of Accesses for this.

4 Results

The results of this work are assessed using three criteria: the speedup of the run-time of the generated application, the time to generate the application and node counts in the AST during the generation process. The first is used to evaluate inversions and small system solving at run-time. The second and third are used to evaluate the operations executed at compile-time. Node count measures the difference in addition of nodes to the IR-AST between the default version and the version implemented in this work. This is an indicator of how large the expressions of matrix entries in the AST are.
Other parts of this work can not be judged by quantitative measures but are implemented and possible to use now or prepare for future applications and extensions. They are mostly covered in section Implementation, so in this section, a showcase demonstrates the tasks in action. They can be used and enable future formulations of problems and extensions to ExaSlang and the generator.

4.1 Speedup at Run-time

To compare the run-times of different configurations to the default version, inversion of filled matrices of different sizes is formulated in an L4-program, together with the built-in timer functionality of L4. In the generated application, the spent time between two points of the program is measured by a timer, such as:

```
1 Var bd_6x6 : Matrix<Double, 6 , 6> = { /* ... blockdiagonal matrix */ }
2 Var bd_6x6_inv : Matrix<Double, 6 , 6>
3 repeat 1000 times {
4         startTimer('bd_i_6x6')
5         bd_6x6_inv = inverse(bd_6x6, "shape=blockdiagonal", "block=2")
6         stopTimer(bd_i_6x6)
7 }
8 print('bd_i_6x6: ', getTotalFromTimer('bd_i_6x6') / 1000)
```

Listing 4.1: code for run-time measurements

The timer `bd_i_6x6` is printed to a timer-class in C++, which adds up the time spent between line 4 and 6. In line 8, the sum is retrieved from the timer and averaged. The matrices are each inverted 1000 times to achieve reliable measurements. For run-time, matrices up to size 50 are used. The same procedure is executed for linear systems, where a system matrix and corresponding unknowns and right-hand-side are set up for different sizes. After that, the system is solved repeatedly and an average run-time is calculated. In this way, the two operations can be compared concerning run-time for different shapes of matrices. The C++-application is compiled with g++ using the following flags:

```
−O3 −DNDEBUG −std=c++11 −I
```

Listing 4.2: used compiler flags for g++

The code runs on an Intel Core i7-8550U CPU sequentially and pinned to the processor by likwid-pin. Likwid is a performance-measurement tool by Regionales Rechenzentrum Erlangen with various commands, `likwid-pin` to force a thread being run to completion on one processor being one of them.
Table 1 shows the run-times for filled matrices with different sizes. The sizes are not chosen larger, because the main application is local matrices in `solve locally`-statements, where matrix sizes are restricted. Seconds are abbreviated as s.

matrix size	run-time, filled matrix [s]
6	2.3e-4
9	8.4e-4
15	0.0024
25	0.012
50	0.072

Table 1: Run-time inversion of a single filled matrix.

These run-times are used to evaluate the advantage that comes with exploiting a matrix shape by a specialized algorithm. To do that, the run-times with the specialized algorithm are also measured and the resulting advantage is assessed by setting the time for execution of the specialized algorithm into relation to the time of the general algorithm by $specialized/general$, thus retrieving a relative run-time.

Blockdiagonal Matrix Inversion

The run-time and speedup of blockdiagonal inversion relative to filled inversion is given in table 2. In the left columns, the matrix size and the size of the diagonal blocks is specified.

matrix size	blocksize	run-time, blockdiagonal matrix [s]	relative run-time compared to filled matrix [%]
6	2	3.7e-6	1.6
6	3	4.11e-6	1.7
9	3	4.2e-6	0.5
15	3	4.6e-6	0.18
15	5	5.97e-4	24.8
25	5	0.00161	13.3
50	5	0.0139	19.0

Table 2: Run-time inversion of a blockdiagonal matrix.

Generally, very large speedups are achieved, ranging from below one percent of the corresponding filled inversion to 24 percent. Furthermore, the larger the diagonal blocks are, the larger the sub matrices to invert, and therefore the smaller the speedup.

Schur-Complement Inversion

To inspect the Schur-inversion's advantages, Schur-matrices with blockdiagonal shape of the A-block are run, whichs results are listed in table 3. Schur-inversion performs worse than blockdiagonal inversion, due to the presence of the B and C-blocks. Although, it performs by far better than filled-inversion for larger matrices. Speedups ranging from under one percent of the run-time of the corresponding filled inversion to 40 percent. Larger B and C-blocks lead to a stronger run-time increase than larger A diagonal blocks, as can be seen when comparing the last two rows of table 3. Larger blocksizes of the A-block increase the run-time. This can be seen when comparing inversion of 50×50 with blocksize of $A = 4$ and blocksize of $A = 6$

Small Linear Systems of Equations

When solving small, filled linear systems directly by LU-decomposition and forward-backward-substitution, instead of calculating an inverse, for 100×100 system matrices a by 5% reduced run-time shows. For larger system sizes, this percentage increases, conforming to the better complexity of solving directly. For smaller matrices, the difference is also not very large, because the matrix multiplication to solve a system by $A^{-1}f$ is always done at compile-time. This reduces the run-time of the solution by inversion and matrix-vector-multiplication to the run-time of a matrix inversion. Solving small linear systems with blockdiagonal (e.g. 15×15 with diagonal block size of 3) or Schur-shapes (e.g. 16×16 with block size of $D = 1$ and block size of the diagonal blocks in the A-block of 3), by applying an optimized algorithm, which exploits the shape of the system matrix also leads to speedups. A run-time of 22 percent and 0.27 percent in comparison to filled systems is achieved. The matrix decomposition executed when solving systems with Schur-shaped system matrix achieves large run-time reductions due to the fact that the matrix-mulitplications involved in the Schur-complement are already executed at compile-time and only very small matrices have to be inverted. The suprisingly low runtime-reduction for solution of blockdiagonal systems in comparison

size	blocksize A-block	blocksize D-block	run-time, blockdiagonal matrix [s]	relative run-time compared to filled matrix [%]
9	2	1	5.8e-6	0.7
9	4	1	3.6e-4	42.4
15	2	1	4.9e-6	0.2
25	4	1	0.00183	15.0
25	6	1	0.0019	16.1
50	4	2	0.0058	8.4
50	7	1	0.0074	10.2
50	6	2	0.0077	10.7

Table 3: Run-time of Schur-inversion with different configurations

to the advantage in run-time for an identically sized inversion (row 5, table 1), assumingly is caused by its implementation, which uses helper-methods to separate the blockdiagonal system under involvement of relatively many copy operations.

4.2 Number of Nodes of Compile-time Operations

ExaStencils possesses a run-configuration, which prints the number of nodes the AST currently consists of after every applied transformation and strategy. This is done by another transformation that traverses the AST and just matches every node, thereby counting all nodes. Using this configuration, the number of nodes added by each transformation can be determined. To determine the number of nodes e.g. the transformation which resolves the `IR_ResolvableMNode`-inversion-nodes adds to the AST, an L4 program only containing the inversion-call and input matrices is run and the node counts are extracted from the output of the generator. Then, the difference between the counts from the AST containing the input matrix and the AST containing the input matrix after compile-time inversion is calculated.
How much of the compile-time is spent for the inversion depends on how many inversions are executed for the application, as well as on how many other operations are executed. One could measure the time, only the inversion takes. But then, the impact on the compile-time duration of large numbers of nodes produced by the inversion is not taken into account. This impact consists of longer traversal times for subsequent transformations. One could also measure the compile-time duration for exemplary applications with lots of inversions, however there are not applications available for many of the variants, Schur and block-diagonal shapes can take, yet. Therefore, inversions and solving of linear systems at compile-time is evaluated by the number of nodes they produce and by the impact they have on the compile-time duration of a test application, in which the operation is executed 10 times. In this way, the impact of larger ASTs for the later stages of the compilation process plays out more than only executing the operation once and measuring the compile-time, because of later inversion strategies long traversal times for larger ASTs resulting from earlier operations. The differences between inversions of small matrices become apparent better with more inversions, too.

4.2.1 Matrices of Accesses

The following node counts are strictly calculated from inversion and solving of linear systems, where every entry is a variable access. That means, none of the entries can be simplified via `IR_GeneralSimplify`, and all arithmetic operations remain being represented as a node. The case of a matrix only consisting of constant expressions and a combination is discussed in section Constant Matrices.

LU vs Cofactor Inversion for Matrices of Accesses

Table 4 shows the number of nodes added to the AST by inverting matrices with no particular shape at compile-time. Note that these node counts are only produced if the whole matrix consists of non-constant expressions. Up to size 3, the direct formulas of Cramer's rule are used. Then, either LU-decomposition or calculating the inverse by building a matrix of cofactors the following definition 2.2 is used. At compile-time only inversions of matrices up to size 6 are comparable to the inversion of a filled matrix, because the latter is only feasible up to this size if no entries can be simplified. Compilation processes including compile-time inversion of larger matrices lead to stack overflows or extremely long compilation times.

matrix size	count of added nodes direct formulas	
2	116	
3	996	
	LU	cofactors
4	28 614	9 382
5	293 928	65 655
6	3 802 681	546 118

Table 4: Node additions to the AST by inversion of a single filled matrix with different methods.

The node count for inversion by LU-decomposition is larger than for cofactor-inversion by a factor of 3-5 for the same matrix size. This is a consequence of the lower complexity of the second method for matrices up to size 5. Due to that, LU-decomposition is not used as the standard method to invert matrices up to this size at compile-time, if it is given as variable accesses. Additionally, the pivoting-problem of section Problems when Pivoting at Compile-time is avoided by this because an inversion with the cofactor matrix does not depend on pivoting. It is not entirely the complexity of the methods, that leads to these large differences in node count: at size 6, an LU-decomposition based inversion should be cheaper than inversion by calculating the cofactor-matrix in terms of complexity: $8/3 * 6^3 = 576$ flops for LU-based inversion and $6! = 720$ flops for cofactor-inversion. However it produces so many nodes, that a compilation containing one of such does not terminate in reasonable time, while cofactor-inversion needs some time, but terminates. This is assumingly caused by the repeated row-updates of LU-factorization.
The increased node count impacts the compilation durations as can be seen in table 5. Seconds are abbreviated as s and matrices are inverted 10 times.

matrix size	compile-time duration	
	LU	cofactors
4	90.1s	27.1s

Table 5: Compile-times of inversion of a filled matrix with different methods.

The node count has a strong impact on the compilation duration.
In table 6, the effects of the optimization described in section Pivot-Elimination-Optimization for LU-Decomposition are displayed. On the right column, the relative amount of nodes compared to the other two inversions for filled matrices is displayed.

size	node count	relative node-count to default LU [%]	relative node-count to cofactors [%]
6	157 609	4.1	29.0

Table 6: Node counts of LU-based inversion with elimination of the pivot expressions.

As this optimization reduces the node count, larger matrices than with the default LU can be decomposed. An inversion this way also is cheaper than inverting by cofactors, as can be seen from the node counts of table 6 and table 4 (last row). If the matrix completely consists of constants, it should not be used, as it introduced accesses, which block the simplification of constant expressions to one resulting constant expression by `IR_SimplifyExpression`. Furthermore, more memory has to be allocated for the variables saving the pivot-elements.

Blockdiagonal Inversion

Table 7 shows the node counts for block-diagonal inversion of suitable sizes, for which diagonal blocks can be set up. As evident, this inversion produces a lot fewer nodes than an inversion of a filled matrix. For example, inverting a 4×4 matrix with 2×2 diagonal blocks produces only 2 percent of the nodes produced when inverting a filled matrix of same size. Therefore, block-diagonal inversion also can be applied to larger matrices, assuming they have a suitable shape, without causing non-terminating compilation. Note, how multiple inversions for matrices of the sizes of the block in table 7 approximately produce an equal number of nodes as a block diagonal inversion with as many of such blocks. E.g. two inversions of 3×3 matrices approximately produce as many nodes as the inversion of a 6×6 matrix with two 3×3 diagonal blocks. This obviously stems from the approach of the block-diagonal inversion-algorithm. In the fourth column, the compile-time duration compared to the inversion of a filled matrix of the same size is listed. The blocks are

size	blocksize	count of added nodes	compile-time duration
2	1	2	3.8s
3	1	4	3.8s
4	1	6	4.1s
4	2	214	4.1s
6	2	322	4.3s
6	3	1 994	6s
8	2	430	4.9s
8	4	24 483	20s
9	3	2 992	6.9s
15	3	8 186	9.2s
15	5	97 336	82.7s (LU with optimization)

Table 7: Node additions to the AST by block-diagonal inversion.

inverted by cofactor inversion if larger than 3×3.

Schur-Complement Inversion

In table 8, the node additions for Schur-inversions for different block sizes are given. The size of the D-block is set to 1 or 2 and the B and C-blocks are nonzero. Furthermore, the A-block has (block)diagonal shape for all tests. The first column corresponds to the matrix size, the second to the size of the diagonal blocks, and the third to the size of the D-block, which at the same time determines the width and height of the B and C-blocks.

Generally, the Schur-inversions produce more nodes than a block-diagonal inversion of the same size, with equal size of the diagonal blocks for the A-block and the diagonal blocks for the block-diagonal matrix. It can be seen that the larger the D-block size is, the more nodes are produced, and therefore more calculation has to be done. This is e.g. evident when comparing rows 2 and 3, which list results for 5×5 matrices. The difference to node counts of comparable block-diagonal inversions is caused by having to apply a more complicated inversion algorithm due to the nonzero entries in the B and C-blocks, which include multiple matrix-matrix-multiplications of submatrices.

size	blocksize A-block	blocksize D-block	count of added nodes	count of added nodes[1]	compile-time duration	compile-time duration[1]
5	2	1	15 719	1 043	19.8s	9.7s
5	1	2	51 427	1 678	73.8s	9.9s
7	2	1	51 189	2 368	80.4s	10.1s
7	3	1	155 765	5 130	-	10.3s
9	2	1	166 869	5 516	-	10.6s
9	4	1	2 408 469	26 429	-	25.6s
16	3	1	2 997 113	24 319	-	21.9s
16	5	1	-	279 101	-	-

Table 8: Node additions to the AST by Schur inversion.
[1] with SchurWithHelpers-optimization

Additionally, the larger the D-block-size is, the more calculation has to be done to multiply the matrices.
Some other results are remarkable: With matrices of size 5 it can be seen, that making the D-block, as well as B and C-block larger has more impact on the computational costs than having larger diagonal blocks. Furthermore, the small difference in shape between a 6×6 blockdiagonal matrix with two 3×3 blocks (row 7 of table 7) and a 7×7 schur matrix with 3×3 blocks and a border set by a D-block of size 1×1 (row 5, table 8) makes for a large difference in node count.
Overall, compile-time inversion by Schur-complement produces far fewer nodes than an inversion of a filled matrix depending on blocksizes and due to that makes the computation of the inverse for these configurations less expensive, as well as shorter for compile-time.
The optimization described in section SchurWithHelpers-optimization performs even better in terms of node-counts, due to saving matrices calculated on the way to separate variables, reducing expression length strongly (right-most columns of table 8). The node count of the optimization is calculated by summing up the node counts of all helpers as well as the resulting inverse. The optimization leads to increased storage demand for the helper variables. This however does not weight heavily for the use case of an inversion before a `solve locally`-loop, because here helper matrices of a small matrix have to be stored once for each loop. It also leads to less cache locality, because instead of accessing values from a continuously stored stack array, values from multiple arrays, eventually saved in different locations, are accessed to evaluate the inverse-expression. In run-time measurements comparing calculation with compile-time-inverted matrices involving and not involving the optimization, this effect does not appear. This is presumably the case due to the small size of the matrices.

Small Linear Systems of Equations

Even though the LU-based inversion performs worse than the cofactor-based inversion on matrices of accesses, small linear systems are more effectively solved at compile-time by using LU-decomposition and forward-backward substitution than inverting per cofactors and multiplying with the right-hand-side. The two modes are compared in table 9. Due to that fact, linear systems can be solved directly up to larger sizes by the first method, than by the second.
When solving systems with blockdiagonal,- or Schur-shaped system matrix, similarily better results are achieved as for inverting a matrix of such shape compared to inverting a filled matrix. This can be seen in table 10, where the node counts for similar configurations of Schur,- and blockdiagonal systems are listed. The first two rows list the configuration for a blockdiagonal shape, the 3rd and 4th row for a Schur-shape. The right-most column reports the difference to solving a system with a general, filled system matrix by LU-decomposition in percent.
When comparing them to the configurations of the same size in table 9, a strongly reduced node count is observed.

size	LU and substitutions	cofactor inverse * f
3	251	996
4	1 100	8 510
5	4 589	64 648
6	168 674	546 118
7	1 317 349	-

Table 9: Node additions when solving small linear systems.

size	blocksize		count of added nodes	relative node count to filled [%]
6	3		535	0.32
size	blocksize A-block	blocksize D-block		
7	3	1	2 552	0.19

Table 10: Node additions to the AST by solution of a small linear system by shape exploiting methods.

Conclusion for Matrices of Accesses

The fact that entries are not known at compile-time makes executing the theoretically more efficient algorithm to solve linear systems and invert matrices, i.e. LU-decomposition, hard to implement. Pivoting becomes impossible due to the lack of information about exact values. Entries can not be simplified meaningfully, and for not completely determined reasons, even at sizes for which the LU-decomposition is in theory computationally cheaper, larger expressions than for cofactor inversion arise. This can be mitigated by the pivot-elimination-optimization. However, if pivoting can not be applied successfully, the algorithm is not applicable. Attempts are made to overcome the pivoting problem as described in section Problems when Pivoting at Compile-time. They show errors made in the LU-decomposition at compile-time at run-time, but this is not constructive for further actions, as it only identifies that the chosen pivots were too small, but not which other pivots to choose. This leads to the following conclusion: the LU-decomposition is useful for cases in which matrices are constant, as described in the next section. Then its superior efficiency shows. It should also be used for inversion and solution of small linear systems at run-time. In the case of matrices of accesses to be decomposed at compile-time, another decomposition technique should be used. The QR-decomposition factorizes a matrix A to a orthogonal matrix Q and a upper triangular matrix R. Similar to LU-decomposition, these factors can be used to solve linear systems of the form $Ax = b$ as follows:

$$\begin{aligned} Ax = QRx &= b \\ Qy &= b \\ Rx &= y \end{aligned} \tag{4.1}$$

The second equation can be solved for y by transposing Q, using its property of an orthogonal matrix. The third equation can be solved for x by backwards-substitution.
QR-decomposition using Householder-reflections requires $4/3n^3$ flops to execute, and is therefore more computationally expensive than LU-decomposition, but cheaper than cofactor-inversion for middle sized matrices in the context of this work [10]. It does not require pivoting, and is due to that the method of choice for cases of matrices filled with accesses. A rudimentary version of it for compile-time execution is implemented in this work. It produced correct results when solving systems in the described way, also for systems where the LU-decomposition fails due to the lack of pivoting. In terms of node counts however, the implemented version showed strong node count production, caused by the calculation of long expressions of norms and subsequent division of

the householder vectors by them. This can possibly be mitigated by eliminating norm expressions similar to eliminating pivot-expressions in LU-decomposition. This is also implemented in this work, as well as two different ways to implement the calculation of the householder matrices. However it not tested and evaluated further than to the point of solving a system correctly.

4.2.2 Constant Matrices

In the previous state of the generator, for constant matrices, cofactor inversion is applied and the resulting matrix is simplified by `IR_GeneralSimplify`, leaving many long expressions up still.
Operating on constant-expression-only matrices makes it possible to exploit two advantages: calculated entries can be simplified, and entry values can be evaluated, so pivoting can be applied. For larger, constant matrices, cofactor-inversion is not considered anymore, because it loses its advantage of not having to pivot as pivoting is now possible effectively. Additionally, it has a higher computational complexity than LU-decomposition based inversion.
In this section, a version of LU-inversion is considered, in which the simplification of constant expressions is done within the computation, simplifying expressions directly when they are produced. The concurrent simplification of expressions is implemented for LU-decomposition in this work. LU loses its disadvantage of creating larger expressions recognized in section LU vs Cofactor Inversion for Matrices of Accesses because the expressions are simplified as they arise. Node counts can not be used as a criterium here anymore, because the node count of the inverted matrix is the same as of the input matrix. The effect of a larger AST also is not present anymore due to that. Furthermore, if entries are simplified directly within the calculation of the LU-decomposition, larger matrices can be inverted. With this method, a matrix of size 20×20 is inverted 10 times at compile-time in 13 seconds. Also, Schur and block-diagonal-matrices can be inverted up to large sizes, because expressions can be simplified here, too. Besides, the main calculations are based on filled inversion. The optimization of simplifying expressions within the computation can also be applied to linear system solving, yielding similar results.
After a successful check, which makes sure that every entry of the matrix is a constant expression, LU-decomposition is applied to invert matrices of sizes larger than 5.
There is the case of matrices, which possess accesses but also constant expressions. In this case, expressions can only be simplified in parts by `IR_GeneralSimplify`. `IR_SimplifyExpression` fails to simplify most expressions, because variable accesses in the argument matrix are quickly propagated to all expressions of the result matrix by LU-decomposition. Expressions containing variable accesses can not be simplified by `IR_SimplifyExpression`. Therefore, the expressions grow, even if small numbers of variable accesses occur in the argument matrix. Inversion of a 7×7 matrix with accesses on the diagonal and constant expressions elsewhere produces enough nodes to slow down the compilation process to nearly a minute.

4.3 Showcases

Shape Classification

To demonstrate the shape classification, the ExaStencils example "2D_FE_Stokes" is considered, which generates a finite element (FE) discretization by Taylor-Hood elements for a Stokes-problem and solves it by solving small local linear systems at runtime of the generated application. A local system, consisting of pressure and velocities in two directions, which are localized at different points of the FE-triangle grid, is set up. The system matrix is of size 19×19. A Schur-shape with block size of $D = 1$, and block size of $A = 9$ is given. It is depicted in figure 4.1 and recognized automatically as well as correctly. Exploiting the shape leads to a run-time of 32.4 percent of the run-time of the default application, produced by the old version of the generator.

Bracket Access

The bracket access is used in example "2D_FV_SWE", a finite-volume-discretized shallow-water-equation problem. In this example, operations to obtain a solution are executed on three different fields `h`,`hu` and `hv`, holding the unknowns of the problem. They can now also be declared as one field with a 3×1 matrix as inner datatype, where one matrix holds the three unknowns for one gridpoint. The unknowns are vectorized this way. Then, operations of three fields become operations of one

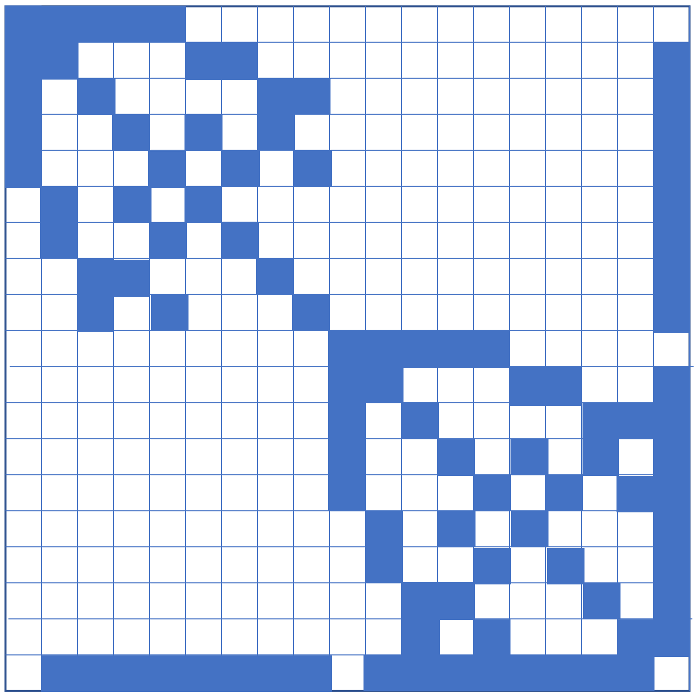


Figure 4.1: Local system matrix of the "2D_FE_Stokes"-problem.

field, accessing different entries of the inner matrices. The original example updates the fields in the following manner:

```
loop over h {
h<next>  =
         hc -
         f*(F0@east - F0@west) -
         f*(G0@north - G0@south)
}
loop over hu {
hu<next> =
         huc +
         f*(S1@east - S1@west) -
         f*(F1@east - F1@west) -
         f*(G1@north - G1@south)
}
loop over hv {
hv<next> =
         hvc +
         f*(S2@north - S2@south) -
         f*(F2@east - F2@west) -
         f*(G2@north - G2@south)
}
```

Listing 4.3: Update of unknowns-fields in the 2D_FV_SWE-example

The fields `h`, `hu` and `hv` are updated in separate loops, each with a similar expression, which are partly set up before the loops in the example (`F0`, `F1`, `S0` ...). `f` denotes a constant factor. `<next>` advices the generator to write the update values to the next slot of the field.

With a vectorized unknowns field `hVec`, combining `h`, `hu` and `hv`, the same operation looks as follows:

```
loop over hVec {
        hVec<next>[0]  = // ...
        hVec<next>[1] = // ...
        hVec<next>[2] = // ...
}
```

Listing 4.4: Update of vectorized unknowns-field in the SWE-example

In line 2, the values previously stores in field `h` are accessed, in line 3 the values of `hu` and in line 4 the values of `hv` are accessed. The statements in the loop are executed for every gridpoint of `hVec`. In this way, only one loop over one field is necessary and the whole example works by operating on one field instead of three.
To furthermore shorten the update, vector additions can be applied: corresponding expressions in all three updates are summarized in the vectors `h`, `S`, `F` and `G` before the loops. E.g. for the summands `h`, `huc`, `hvc`, as well as the `S0`, `S1` and `S2`:

```
Expr h = {{h}, {huc}, {hvc}}
Expr S = {{0},{S1@east  - S1@west},{S2@north - S2@south }}
```

Listing 4.5: Setup of vector expressions for the update

One entry of the field `hVec` is then updated by:

```
loop over hVec {
        hVec<next>[:] = = h + f * S - f * F - f * G
}
```

Listing 4.6: Update by vector-addition

This updates the former `h`, `hu` and `hv` by one assignment, adding a expression of vectors to one entry of the matrix-field `hVec`. The update is now more conforming to a mathematical formulation of it and shorter as well as more simple, making it less error prone.

5 Conclusion

In this work, five interconnected tasks were treated.
The process of resolving matrices in the generator was refactored. Specialized AST-nodes were set up for matrix operations. They possess specific attributes conforming to the characteristics of the different matrix operations and provide convenience and encapsulation for the information connected to each operation. The nodes were grouped by extending traits, each trait abstracting an AST-operation to be executed on them. The AST-operations were: (i) extraction from function calls, (ii) processing of operations that can be executed at run,- and at compile-time and (iii) resolving the operation by calculating a result and inserting it into the AST. In this way, AST strategies could be formulated by addressing nodes of the trait type. More concise, more extendable, and more generally formulated strategies were achieved.
The DSL ExaSlang was extended by the slice-functions `getSlice` and `setSlice`, implementing slicing of variable size. Moreover, access and slicing of matrices by the bracket operator were implemented. This took place at different stages of the ExaStencils generation process: a parser was added to the L4-parser. AST-nodes to resolve bracket-operations on two types of objects, plain matrix variables, and matrix fields, were set up. They are used to pick accesses to the two object types at respectively convenient points in the generation process, for plain matrix variables on layer 4, for matrix fields on IR. At last, they map bracket-slicing and bracket-access to the implemented slicing functions `getSlice` and `setSlice`, or access functions. Slicing matrices at compile-time and run-time was implemented and tested in various scenarios successfully, except for slices of run-time dependent size. This leads to the conclusion, that matrix datatypes, which are allocated at run-time of the application, would be useful for such cases, although they have to be carefully implemented to not reduce the performance of the application by time consuming system calls.
Matrix shapes can be exploited to strongly reduce the computational effort for operations like inversion and the solution of small linear systems of equations. Parser support and specialized AST-nodes, `L4/IR_MatShape`, were implemented to transport information about matrix shapes in ExaSlang L4-programs, in the L4-AST, and the IR-AST. The design of the nodes aims at extendability for more matrix shapes than targeted in this work. Furthermore, a classification algorithm for Schur,- and block-diagonal shapes was set up, successfully recognizing them in test applications real-world applications alike. A matrix shape can not be classified if the matrix only consists of variable accesses.
A collector was implemented, to provide constant expressions for the classification of matrix shapes and to enable certain operations involved in matrix inversion and linear system solution.
Inversion by LU-decomposition was formulated for compile-time and run-time execution. It unexpectedly produced higher AST-node counts than the inversion by cofactors, if executed on a matrix filled with variable accesses at compile-time. Additionally, the necessary pivoting is impossible for matrices filled with accesses. The collector therefore tries to provide a constant initial expression. If this is not possible due to write-accesses to the matrix or the lack of an initialization expression, the method is not applicable. If executed on a constant or partly constant matrix with a parallel simplification of expressions, it performed better than the priorly implemented methods, inverting filled matrices up to size 20 and more in seconds. Here, pivoting is possible, too. It outperforms the Cofactor inversion, which was the default method for this case, due to its lower complexity and due to the built-in mechanism to simplify expressions during execution.
Inversion algorithms that exploit the shape of the matrix were implemented for compile-time and run-time execution. Strongly reduced node counts in the AST and speedups at run-time were achieved.
A blockdiagonal matrix of size 25×25 with 5×5 blocks was inverted running for 13.3 percent of the run-time of inversion of a filled matrix of identical size. For Schur-matrices similar speedups are achieved, e.g. 15 percent run-time for a 25×25 Schur-matrix with 6×6 diagonal blocks.
For compile-time inversion: Matrices of accesses up to size 15 with inner blocks of size 3 can be inverted 10 times in about 10 seconds with the new implementation, provided they are of a Schur or block-diagonal shape. For Schur-shapes, increasing block size of the D-block increases the computational effort to invert them rapidly. For compile-time execution of the Schur-inversion, additionally an optimization was implemented that eliminates common subexpression, reducing node counts strongly: in a configuration of a 16×16 Schur-matrix with 3×3 blocks 0.8 percent of the node production of a default version were produced. There, the expressions of the component matrices

forming the Schur-complement were targeted.
At last, the direct solution of linear systems by LU-decomposition and forward/backward substitution was set up. It can be executed at run-time and compile-time. The options to use the functionality for developers of the framework as well as L4-users were provided, by setting up the new AST-nodes `L4/IR_SolveMatSys` and extending the L4-parser. At run-time, only small speedups were achieved in comparison to solving a system by inversion up to size 50 of the system matrix. At compile-time, a 70 percent reduced node count was achieved in comparison to a solution of linear systems by cofactor-inversion at size 6.
Moreover, also for linear system solution optimizations based on the matrix shape were implemented for block-diagonal and Schur-shapes. Exploiting the shape of a blockdiagonal linear system of equations produces only 12 percent of the nodes produced without exploitation of the shape during compile-time execution. At run-time, speedups of e.g. 22 percent for blockdiagonal systems resulted.

6 Future Work

When inverting a matrix as well as solving linear systems besides block-diagonal and Schur-shapes, there are other shapes a matrix can have. These can be exploited to achieve shorter run-time or less compile-time, too. E.g. for Band-matrices with bandwidth 3, the *Thomas-algorithm* can be employed to solve a linear system with only n operations. For other bandwidths, a *band LU-decomposition* can be employed, also reducing the computational cost drastically in comparison to a default LU-decomposition. Furthermore, for symmetric positive-definite matrices, a *Cholesky-decomposition* can be used to solve systems. The matrix-classification can be extended for such shapes by e.g. checks for bands in a matrix to also recognize them and produce a `matShape`-object, which stores the individual attributes, for instance, bandwidth. Further cases can be added to inversion and solving of linear systems classes, which branch to the specialized algorithm corresponding to the shape.

The QR-decomposition can be used to replace the LU-decomposition as a method so solve linear systems of equations. Its advantage over LU-decomposition is that it does not require pivoting and can therefore be applied to matrices completely consisting of accesses. In the current naive implementation, it produces more nodes than the inversion by Cofactors. Eliminating expressions in the algorithm similar to the pivot-elimination optimization for LU-decomposition might change this.

If write accesses to a matrix only write to a single entry of the matrix, currently the matrix is still considered as completely non constant by the collector which retrieves constant initialization expressions. It returns `None` then. But most entries of the matrix are in fact constant and their value can be retrieved. In the future, the collector can be adapted to also save the position of the entry that is written to. It then can retrieve a matrix consisting of constant values except for the entry written to, which is replaced by a variable access. This expression can afterwards be used by the classification and solve algorithms.

Parsing `solve locally`-statements can be extended to also offer an interface for matrix shapes for the local systems, in the same way shapes can be passed to `solveMatSys`-statements.

Users of the generator can be roughly categorized into two sections, regarding the state of the development of their applications: some are still making frequent changes to their ExaSlang application, e.g. adding functions or adapting parameters. Their application is in an early state. Therefore, they often have to newly generate code to see the impact and results of their modifications. Conclusively, their need is not yet a in every regard highly optimized application with minimal run-time, but a short compile-time, to not have to wait long between a modification and the viewing of its result. Other users are at the end of the development of a test case. They would like their solver to be as fast as possible to use it and do not care about a longer compile-time, as its only once. From the viewpoint of the first category of users, computationally expensive operations can be shifted to run-time, if a shorter compile-time is achieved, to make quick development of their application possible. The second group of users want everything that can be done at compile-time

executed at compile-time to minimize run-time of the generated application, also accepting a longer time to generate the application.
Currently, a user has to make many smaller decisions by adapting knowledge attributes and adapting settings of the generator, which influence the duration of run,- and compile-time. Most of the time, these aim at one of the two goals. An automatization of the implementation of such priorities might look as follows: A user only provides the priority he has and the generator automatically makes all smaller decisions in the favor of it, under use of the knowledge of an expert of the generator, who knows well how setting certain switches impacts run-, and compile-time. The generator e.g. inverts matrices which are small enough to not matter for the compile-time duration at compile-time, but shifts all inversion of meaningfully large matrices to run-time, to reduce the compile-time as much as possible.

References

[1] Alfred V. Aho et al. *Compiler*. 2008.

[2] Siegfried Bosch. *Lineare Algebra*. 2008.

[10] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. 2013.

[11] Sebastian Kuckuk. "Automatic Code Generation for Massively Parallel Applications in Computational Fluid Dynamics". PhD thesis. 2019.

[12] Christian Lengauer et al. "ExaStencils: Advanced Multigrid Solver Generation". In: *Software for Exascale Computing – SPPEXA 2016–2019*. Ed. by Hans-Joachim Bungartz et al. Vol. 136. Lecture Notes in Computational Science and Engineering. Springer, July 2020, pp. 405–452. ISBN: 978-3-030-47955-8. DOI: 10.1007/978-3-030-47956-5_14.

[13] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*.

[14] Martin Fowler with Rebecca Parsons. *Domain Specific Languages*. 2010.

[15] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. 2003.

[16] Christian Schmitt et al. "ExaSlang: A Domain-Specific Language for Highly Scalable Multigrid Solvers". In: *Proc. of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)* (New Orleans, LA, USA). UnivIS-Import:2015-04-16:Pub.2014.tech.IMMD.inform.exasla. New York, NY, USA: IEEE Press, Nov. 17, 2014–Nov. 17, 2014, pp. 42–51. ISBN: 978-1-4799-7020-9. DOI: 10.1109/WOLFHPC.2014.11.

[17] Christian Schmitt et al. "Systems of Partial Differential Equations in ExaSlang". In: *Software for Exascale Computing - SPPEXA 2013-2015*. Ed. by Hans-Joachim Bungartz, Philipp Neumann, and Wolfgang E. Nagel. Cham: Springer International Publishing, 2016, pp. 47–67. ISBN: 978-3-319-40528-5.

[18] Gilbert Strang. *Linear Algebra and Its Applications*. 2006.

[19] Fuzhen Zhang. *The Schur Complement and Its Applications*. 2005.

Code references

[3] *ExaStencils code repository*. URL: https://i10git.cs.fau.de/exastencils/release.

[4] *ExaStencils* `IR_GeneralSimplify`. URL: https://i10git.cs.fau.de/exastencils/release/-/blob/master/Compiler/src/exastencils/optimization/ir/IR_GeneralSimplify.scala.

[5] *ExaStencils* `IR_MatrixAccess` *with resolve-strategies*. URL: https://i10git.cs.fau.de/exastencils/release/-/blob/master/Compiler/src/exastencils/baseExt/ir/IR_MatrixAccess.scala.

[6] *ExaStencils* `IR_SimplifyExpression`. URL: https://i10git.cs.fau.de/exastencils/release/-/blob/master/Compiler/src/exastencils/optimization/ir/IR_SimplifyExpression.scala.

[7] *ExaStencils* `L4_MatrixAccess`. URL: https://i10git.cs.fau.de/exastencils/release/-/blob/master/Compiler/src/exastencils/baseExt/l4/L4_MatrixAccess.scala.

[8] *ExaStencils* `L4_Parser`. URL: https://i10git.cs.fau.de/exastencils/release/-/blob/master/Compiler/src/exastencils/parsers/l4/L4_Parser.scala.

[9] *ExaStencils* `L4_VariableDeclarationCollector`. URL: https://i10git.cs.fau.de/exastencils/release/-/blob/master/Compiler/src/exastencils/util/l4/L4_VariableDeclarationCollector.scala.