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

Lehrstuhl für Informatik 10 (Systemsimulation)

Extension of the ExaStencils Framework with Tensors

Martin Zeus

Bachelor Thesis

Extension of the ExaStencils Framework with Tensors

Martin Zeus

Bachelor Thesis

Aufgabensteller:	Prof. Dr. H. Köstler
Betreuer:	Dr. Ing. S. Kuckuk
Bearbeitungszeitraum:	1.4.2020 – 30.09.2020

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 Thesis einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 31. Oktober 2020 .....

Contents

1 Abstract

The field of system simulation is growing continuously. This matter is comprehensible to the extent that the size of problems, especially in physics and technology, is increasing. To tackle the resulting software demand, frameworks for code generation were developed. As an example, ExaStencils [Kuc19], which can generate massive parallel C++ Code out of ExaSlang [Sch+14], a domain-specific script language. The framework transcends in 4 steps the linguistic gap between formal mathematical definitions and fast hardware near code. Each step is adjusted to the needs of a single user class. ExaStencils based on the functional language Scala [Sca].

Mr. Kuckuk described in his work [Kuc19] how ExaSlang can be transformed into a processor near and faster language (e.g. C++). The doctoral thesis also shows which benefits are included in this category of tools for scientific calculation, especially for numerical solvers who find the solution of partial differential equations [Noac] (e.g. Navier-stokes-equation [Tem01])

ExaStencils has a four-layer approach, which means that each layer transfers the code peu à peu from an abstract mathematical description to a technical view. A fifth layer, named IR, represents the concrete implementation. This implements a print function, which writes the transformed layers into correct target code.

Tensors are a complex mathematical data structure related to a matrix. Theoretically, it assigns a vector- or matrix- associated structure to a coordinate space. The main goal of the thesis was to implement those tensors to ExaStencils. It approaches the problem from the short theoretical view over the fourth layer to the implementation. This work defines tensor and its operations, whereas, for proofs, it is referencing the literature. The thesis discusses where the type is used and how it can be represented and defined in ExaSlang. Concertedly with the ExaSlang part comes the definition of the parser rules and further the implementation in the fourth layer.At this part, the biggest changes for those complex datatype are possible in the context of code generation. Hence, one of the work was to seek for achievements of unique tensors in ExaStencils.

The first achievement was to improve the usability of complex data structures. The thesis gives an insight into how modern access methods are implementable in ExaStencils. Further, it describes the implementation of operations on tensors, notably that there was a strong focus on using compilation time algorithmic instead of increasing the runtime in this connection.

The Einstein notation played a significant role in this task. In numerous literature, this abbreviated notation is used to formulate sentences with tensors. One effort was to implement the notation in the IR-layer.

As a result, it has been shown that it is possible to integrate such complex data type into ExaStencils without dangling the runtime and improving the usability of the tool itself.

2 Introduction

Many physical and technical issues are described with tensors. For example, continuous mechanics is a wide used formalism, especially in the theory of elasticity (strain tensor), which uses tensors. Other applications can be found widely, for instance in electrodynamics (electromagnetic field tensor).

Tensors describe the relationship between a vector- or matrix-related data structure and its space. Usually used spaces are classical coordinate spaces in R, e.g., the cartesian vector space in

N dimensions, but more non-classical spaces comparable to topological spaces are possible. The code and thesis only include spaces that store numerical data types in array-like storages related to Euclidean space. Tensors can easily describe points or objects in an ecosystem because of the tensor values' inherent bounding with the coordinate system. For this reason, they are commonly used to represent locations in a grid. As an instance, in the theory of elasticity, they are used to calculate the deformation of the mesh if the temperature has changed. The examples part gives herefore a more concrete insight.

ExaSlang is a domain-specific language (apr. DSL) for high-performance computing related to the known language Julia [Noae]. DSLs are scripting or programming languages, which have a concrete scope of application, for instance, high-performance computing. In contrast, C++, a classical programming language, is used for a broad field. Possible use-cases are desktop applications, computer games, embedded solutions, or also scientific computing. DSLs' development is driven by the problem that classical languages often work not optimal enough in specific fields or need expensive development effort or libraries. Distinctive to other languages, ExaSlang is not constructed to be compiled to byte code. ExaStencils parses ExaSlang and generates different target code, depending on the target system. Such target code can be C++ to run on CPUs, or also OpenGL is possible if the calculation has to be done at graphic processors. As a difference between code-generation technology and classical software development is where intelligence and functionality are based. When the software is directly written in the target language, every called method, data types, and libraries are also in the same language. The compiler transfers the whole project together into byte code. That differs from ExaStencils, where the application development is in ExaSlang and compiled to source code again. In the moment of parsing the ExaSlang-code, the functionality which is implemented in the code-generator directly can be used. So, operations on the graph and complex data types can be done before printing and later compiling the target code. This procedural manner is an influential possibility to reduce calculation effort on the target system because necessary operations for the application can be moved to the generation time and consequently to away from runtime.

Runtime on a high-performance cluster is expensive, especially for small companies, which are often not able to operate on an own small cluster economical. Only a few concerns have the properties to acquire a high-performance workstation. Supercomputers not only cost money for the purchase, but they also require maintenance and, in particular, energy for working. The issue of energy consumption in IT can also be viewed from an environmental side. To reduce the amount of humans energy consumption, software development should avoid unnecessary computational activity.

As written, ExaSlang and ExaStencils are conceptually tailored for the usage in the field of high-performance computing. Frameworks for scientific calculation implicit needs mathematical data types because models described mathematically. This thesis shows in some small examples where tensors can be found in such models. It also displays why moving operational work to the generation time is worthwhile.

The first part of this work aims to perceive all necessary content a tensor in a code generation framework. The part of the theoretical background is a result of this step. This part is homogenously designed to obtain an easy reading on the one hand and the other a clear and formal correct structure. Definitions follow brief instructions and them again compact examples. This task ends with a short introduction in the usual wording and ground standing functionality of the code generation tool. The later parts, the representation and the implementation, repeat the topics from the theoretical background for fast searching. The representation precise the tensor type and functionality in ExaSlang. It also indicates how tensors can be expressed in a formal language. From now on, there is a limitation. When discussing ExaSlang and also ExaStencils, this thesis only includes the fourth layer of ExaSlang and in ExaStencils only the fourth and the IR layer. The last and also most significant part of this work is the implementation. The main part describes at first the parser and the fourth layer, which includes more precisely how ExaSlang-code will be parsed and how objects will be created out of the source. At the third topic of the implementation, the thesis describes the development on the IR layer. Although the topic starts with the tensor data type and the expression, the main exertion was the transformations. A relevant point is the realization of the functionality, which claims most of the whole working time. The thesis explains the algorithm of Einstein notation extensively. This algorithm and the deduced contraction is one of the most complicated elements of the thesis. After the IR layer, the thesis lists the tests of the

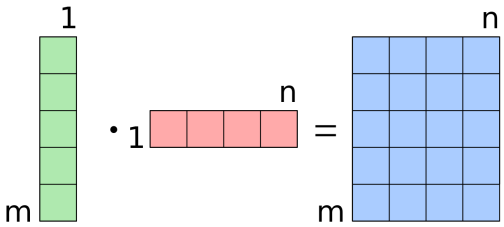

Figure 1: Dyadic product [VQEW18]

tensor class. The last topic has its focus on explicating the results.

3 Theoretical background

The following section introduces the theoretical background of tensors and calculus. It also gives some calculation examples, whereas for proofs it reference to the bibliography. The last part of the theoretical background describes general information to code generation, parsing and ExaStencils.

3.1 Dyadic product

In linear algebra a dyadic product is understood as a product of two vectors.

Definition 1 (Dyadic product)**.** *Are $u \in \mathbb{R}^n, v \in \mathbb{R}^m$ vectors then the dyadic product maps $\otimes : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^{m \times n}, (u, v) \mapsto u \otimes v$. The dyadic product $v \otimes u = uv^T = z$ is defined when $\forall z_{ij}$ applies $z_{ij} = u_i * v_j$ with $i \leq n \wedge j \leq m$.*

Corollary 1.1 (Commutativity)**.** *The dyadic product is not commutative.*

Example:

$$u \in \mathbb{R}^3, v \in \mathbb{R}^4 \to u \otimes v = y \in \mathbb{R}^{3 \times 4}, v \otimes u = z \in \mathbb{R}^{4 \times 3} \Rightarrow y \neq z \tag{1}$$

Corollary 1.2 (Distributivity)**.** *The dyadic product is distributive.*

3.2 Definition Tensor

Note: In the further descriptions of tensors, I have chosen a 3D space because it is the most common usage. In general, tensors can have any dimensionality. Later I talk about the implementation, in which the users can choose the dimensionality on their own.

When the location of points in three-dimensional cartesian space is described, every point is a linear combination of three orthogonal vectors.

$$f : \mathbb{R}^3 \mapsto \mathbb{R}^3, f(x * e_x, y * e_y, z * e_z) \to \phi, \begin{pmatrix} x \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ y \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ z \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{2}$$

Or more exactly a factor on the orthogonal basis. In the 3D Euclidean space is the canonical basis e, where e_n, e_m with $n \neq m$ are orthogonal.

$$e_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, e_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, e_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{3}$$

Now we are able to construct a first-order tensor out of this information. The tensor can be described as the product of the factor with his base.

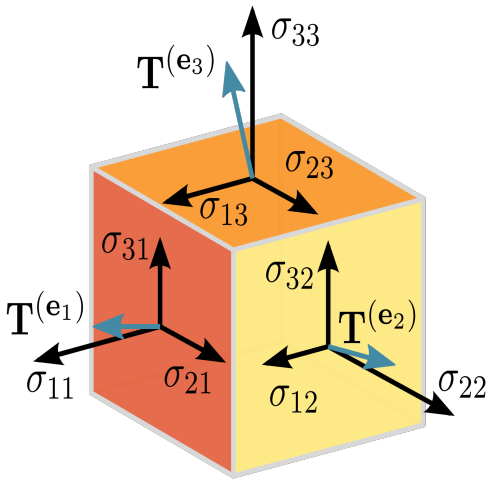

Figure 2: Second order tensor [Fig]

$$a = \phi_i e_i = \sum_{k=1}^{3} \phi_k e_k \tag{4}$$

A further possible step is to give every axis its own base. This happens by creating a second-order tensor with the dyadic product over the first-order tensors.

$$v = a_i e_i, w = b_i f_i \rightarrow \mathcal{T} = v \otimes w = (a_i e_i) \otimes (b_i f_i) \tag{5}$$

$$\mathcal{T} = \begin{pmatrix} v_1 * w_1 & v_1 * w_2 & v_1 * w_3 \\ v_2 * w_1 & v_2 * w_2 & v_2 * w_3 \\ v_3 * w_1 & v_3 * w_2 & v_3 * w_3 \end{pmatrix} = \begin{pmatrix} a_1 * e_1 * b_1 * f_1 & a_1 * e_1 * b_2 * f_2 & a_1 * e_1 * b_3 * f_3 \\ a_2 * e_2 * b_1 * f_1 & a_2 * e_2 * b_2 * f_2 & a_2 * e_2 * b_3 * f_3 \\ a_3 * e_3 * b_1 * f_1 & a_3 * e_3 * b_2 * f_2 & a_3 * e_3 * b_3 * f_3 \end{pmatrix} \tag{6}$$

In general, a Tensor describes the relationship between algebraic objects and their vector space. [Sha04] This is the reason why tensors have different dimension form depending on the space. For example, when a Tensor $\mathcal{A}$ characterizes the dependence between scalar objects, $\mathcal{A}$ also has to be a one dimensional. In the following document, we choose small letters to describe scalars and also scalar tensors. Typical vector spaces are the different coordinate spaces such as cartesian or geographic coordinate space. More generally, it is also possible to use topological spaces, for instance, the Hilbert space. In literature are different versions of the definition of a tensor. Following, I will introduce the mainly used one:

3.2.1 Definition via multilinear map

May $\mathbb{L}^n$ is a set of linear mappings, which transforms a vector into another in the euclidean space $\mathbb{E}^n$.

$$y = Ax, y \in \mathbb{E}^n, \forall x \in \mathbb{E}^n, \forall A \in \mathbb{L}^n \tag{7}$$

Thus elements of $\mathbb{L}^n$ are called second-order tensors. The fourth-order tensor has the same definition, whereas y, x are no vectors but second-order tensors. [Its19]

$$\mathcal{Y} = A : \mathcal{X}, \ \mathcal{A}, \mathcal{Y} \in \mathbb{L}^n, \forall \mathcal{X} \in \mathbb{L}^n \tag{8}$$

3.3 Einstein notation

The Einstein notation is a simplification formulated by Einstein in 1916 [Åh02]. The abbreviation is widely used in tensor calculus, which is why it has to be mentioned here.

Definition 2 (Einstein notation)**.** *May* $\{i \in \mathbb{N}\}$ *limited and*

$$f = \sum_{l \in i} \phi(l)\psi(l)$$

then the term can also be written as $\phi_i\psi_i$*. Where* i *is the domain of the sum.*

Examples:

$$(a_i + b^i) * a_k = \sum_{l=1}^{i} (a_l + b^l) * a_k \tag{9}$$

$$(a_i + b^i) + (c_k * d_k) + e_l = \sum_{m=1}^{i} (a_m + b^m) + \sum_{n=1}^{k} (c_n * d_n) + e_l \tag{10}$$

In the second example, i is not at the index but in the superscript. Important is for the notation only that a collective iteration variable exists, not that a part is indexed related to vectors or matrix. In addition to the classical Einstein notation, the module in ExaStencils will allow more than two components. For instance $t1_i * t2_i * t3_i$ is also allowed.

3.4 Basic arithmetic operations on tensors

The most common calculation rules for tensors with order up to 2 are also defined in common matrix space for their expression.

3.4.1 Multiplication with scalars

Definition 3 (Multiplication tensor with scalar)**.** *May* T, Z *tensors with the same order and dimensionality and* $t \in T, z \in Z$ *elements of the tensors. May also a scalar* $x \in \mathbb{R}$*. Then* $Z = T * x$ *if* $i \in \mathbb{N}$ *is an iterator over all indices of a tensor and* $z_i = t_i * x, \forall i.$ *.*

Example:

$$a \in \mathbb{R}, \mathcal{T} = \begin{pmatrix} t_{11} & t_{21} & t_{31} \\ t_{12} & t_{22} & t_{32} \\ t_{13} & t_{23} & t_{33} \end{pmatrix}, \text{then: } \mathcal{U} = a * T = \begin{pmatrix} a * t_{11} & a * t_{21} & a * t_{31} \\ a * t_{12} & a * t_{22} & a * t_{32} \\ a * t_{13} & a * t_{23} & a * t_{33} \end{pmatrix} \tag{11}$$

3.4.2 Division with scalars

Definition 4 (Division tensor with scalar)**.** *May* T, Z *tensors with the same order and dimensionality and* $t \in T, z \in Z$ *elements of the tensors. May also a scalar* $a \in \mathbb{R}$*. Then* $Z = T/a$ *if* $i \in \mathbb{N}$ *is an iterator over all indices of a tensor and* $z_i = t_i/x, \forall i.$ *.*

Example:

$$a \in \mathbb{R}, \mathcal{T} = \begin{pmatrix} t_{11} & t_{21} & t_{31} \\ t_{12} & t_{22} & t_{32} \\ t_{13} & t_{23} & t_{33} \end{pmatrix}, \text{then: } \mathcal{U} = T/a = \begin{pmatrix} t_{11}/a & t_{21}/a & t_{31}/a \\ t_{12}/a & t_{22}/a & t_{32}/a \\ t_{13}/a & t_{23}/a & t_{33}/a \end{pmatrix} \tag{12}$$

A simple resulting operator is an inversion under a given scalar. If the scalar is 1 than the resultig tensor is the inverse of multiplication.

Definition 5 (Inverse division tensor with scalar)**.** *May* T, Z *tensors with the same order and dimensionality and* $t \in T, z \in Z$ *elements of the tensors. May also a scalar* $a \in \mathbb{R}$*. Then* $Z = a/T$ *if* $i \in \mathbb{N}$ *is an iterator over all indices of a tensor and* $z_i = t_i/x, \forall i.$ *.*

Example:

$$a \in \mathbb{R}, \mathcal{T} = \begin{pmatrix} t_{11} & t_{21} & t_{31} \\ t_{12} & t_{22} & t_{32} \\ t_{13} & t_{23} & t_{33} \end{pmatrix}, \text{then: } \mathcal{U} = a/T = \begin{pmatrix} a/t_{11} & a/t_{21} & a/t_{31} \\ a/t_{12} & a/t_{22} & a/t_{32} \\ a/t_{13} & a/t_{23} & a/t_{33} \end{pmatrix} \tag{13}$$

This functionality is not standard for tensors, and so the definition is made for this thesis. Tensors will mostly be used with matrixes and scalars side by side, so the tensor module should support enough rudimental functionality to build more complex statements. For instance, this function can be used to calculate the inverse of a diagonal tensor.

3.4.3 Addition with vectors/matrix

Both the addition of two vectors/matrix and one of a tensor and an vector/matrix is only defined for those with the same dimensions. Therefore it is only possible to add a n-order tensor to a matrix within $\mathbb{R}^{3^n}$.

Definition 6 (Addition tensor with matrix)**.** *May $A \in \mathbb{R}^{3^n}$ and T, Z tensors with the order n and $a \in A, t \in T, z \in Z$ elements of A, T, Z. Then $Z = T + A$ if $\forall z \in Z : z = t + a$ where a, t, z has the same location in the tensor of matrix.*

Example:

$$A = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}, \mathcal{T} = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix}, \text{then: } \mathcal{Z} = A + T = \begin{pmatrix} a_1 + t_1 \\ a_2 + t_2 \\ a_3 + t_3 \end{pmatrix} \tag{14}$$

3.4.4 Multiplication with vectors/matrix

Both the addition of two vectors/matrix and one of a tensor and a vector/matrix is only defined for those with the same dimensions. Therefore it is only possible to multiply a n-order tensor to a matrix within $\mathbb{R}^{3^n}$.

Definition 7 (Multiplication tensor with matrix)**.** *May $A \in \mathbb{R}^{3^n}$ and T, Z tensors with the order n and $a \in A, t \in T, z \in Z$ elements of A, T, Z. Then $Z = T * A$ if $\forall z \in Z : z = t * a$ where a, t, z has the same location in the tensor of matrix.*

Example:

$$A = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}, \mathcal{T} = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix}, \text{then: } \mathcal{Z} = A * T = \begin{pmatrix} a_1 * t_1 \\ a_2 * t_2 \\ a_3 * t_3 \end{pmatrix} \tag{15}$$

3.4.5 Multiplication/Addition between two tensors

To develop higher functions, or more generally linear transformations, it is necessary to define arithmetic operators. Therefore, I have to note that multiplication is not similar to tensor product. They are basically different concepts. Multiplication in that context means an elementwise operator similar to the Addition.

Definition 8 (Addition two tensors)**.** *May T, U, Z tensors with the same order and dimensionality and $t \in T, u \in U, z \in Z$ elements of the tensors. Then $Z = T + U$ if $i \in \mathbb{N}$ is an iterator over all indices of a tensor and $z_i = t_i + u_i, \forall i$.*

Example:

$$\mathcal{T} = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix}, \mathcal{U} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}, \text{then: } \mathcal{Z} = \mathcal{T} + \mathcal{U} = \begin{pmatrix} t_1 + u_1 \\ t_2 + u_2 \\ t_3 + u_3 \end{pmatrix} \tag{16}$$

Definition 9 (Elementwise multiplication two tensors)**.** *May T, U, Z tensors with the same order and dimensionality and $t \in T, u \in U, z \in Z$ elements of the tensors. Then $Z = T * U$ if $i \in \mathbb{N}$ is an iterator over all indices of a tensor and $z_i = t_i * u_i, \forall i$.*

Example:

$$\mathcal{T} = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix}, \mathcal{U} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}, \text{then: } \mathcal{Z} = \mathcal{T} * \mathcal{U} = \begin{pmatrix} t_1 * u_1 \\ t_2 * u_2 \\ t_3 * u_3 \end{pmatrix} \tag{17}$$

3.5 Algebra of tensors

Tensors are commutative groups in the context of addition [8] and multiplication [9]. [Its19]

May $\mathbb{1}, \Phi, \mathbb{T}, \mathcal{U}$ tensors with order k and dimension n, then $\mathbb{1}$ is the Kronecker Delta [11] with the order $k \in \mathbb{N}$, so $\mathbb{1}$ is the generalization of the identity matrix in the tensor room. $\mathbb{1}$ is the neutral element of multiplication. May also Φ a zero tensor with order $k \in \mathbb{N}$ and dimension n, and α, β scalars. Then applies,

$$\mathbb{T} * \mathbb{1} = \mathbb{1} * \mathbb{T} = \mathbb{T} \tag{18}$$
$$\mathbb{T} + \Phi = \Phi + \mathbb{T} = \mathbb{T} \tag{19}$$
$$\mathbb{T} - \mathbb{T} = \Phi \tag{20}$$
$$\mathbb{T} + \mathbb{U} = \mathbb{U} + \mathbb{T} \tag{21}$$
$$(\mathbb{T} + \mathbb{U}) + \mathbb{V} = \mathbb{T} + (\mathbb{U} + \mathbb{V}) \tag{22}$$
$$\alpha * (\mathbb{T} + \mathbb{U}) = \alpha * \mathbb{T} + \alpha * \mathbb{U} \tag{23}$$
$$\alpha * (\beta * \mathbb{T}) = (\alpha * \beta) * \mathbb{T} \tag{24}$$

3.6 Tensor calculus

3.6.1 Double contraction

Definition 10 (Double contraction)**.** *May $\mathbb{A}$ a tensor with order $4 \vee 2$ and $\mathcal{B}$ second-order tensors, then : is a double Contraction if $\mathbb{A} : B = A_{ijkl}B_{kl}$ and $\mathbb{A}$ has a fourth-order or if $\mathbb{A} : B = A_{kl}B_{kl}$ and $\mathbb{A}$ has a second-order [Hol00].*

For some particular cases the contraction of a fourth-order with a second-order tensor is needed (e.g. elasticity). The syntax of the contraction is similar to Einstein notation. [2].

3.6.2 Kronecker delta

Definition 11 (Kronecker delta)**.** *May δ_{ij} and $i, j \in \mathbb{N}$ a function then it is called Kronecker delta, when*

$$\delta_{ij} = \begin{cases} 0, i \neq j \\ 1, i = j \end{cases}$$

There a reasonable generalisation if for objects with a higher-order correspondingly, more indices are allowed.

3.6.3 Trace

Definition 12 (Trace, 2nd order Tensor)**.** *May $\mathcal{A}$ a second order tensor, then tr is a trace if $trA = A_{ii} = a_{11} + a_{22} + a_{33}$ [Hol00]*

Definition 13 (Trace, n-th order Tensor)**.** *May $\mathbb{A}$ a tensor with order n, then tr is a trace if $trA = \Sigma\left(\delta_{i,..} * \mathbb{A}_{i,...}\right)$ and δ is the Kronecker delta [11] over all index dimension.*

3.6.4 Determinant

Definition 14 (Determinant)**.** *May $\mathcal{A}$ a second order tensor and $A \in \mathbb{R}^{3\times3}$ a matrix of tensor components, then det is a determinant of $\mathcal{A}$ if*

$$det\mathcal{A} = detA = det \begin{bmatrix} \mathcal{A}_{11} & \mathcal{A}_{12} & \mathcal{A}_{13} \\ \mathcal{A}_{21} & \mathcal{A}_{22} & \mathcal{A}_{23} \\ \mathcal{A}_{31} & \mathcal{A}_{32} & \mathcal{A}_{33} \end{bmatrix}$$

3.6.5 Eigenvalues

Considering the workload, I only determine the eigenvalues for tensors with order two. So it is possible to use algorithms for the determination of matrix eigenvalues.

For the calculation of the eigenvalues I chose the QR-algorithm which basically computes an upper Hesseberg-matrix in limited steps. In practical applications it is unusual to have very high dimensional tensors, because of the common 3D-Euclidean-vectorspace. Given that limitation, I choose the basic QR-Algorithm with a single shift and a linear matrix polynomial as the result. [Arb]

Algorithm 1: QR-Algorithm with single shift

Data: Tensor 2nd order $\mathcal{T}$
Result: output data p
$A_0 := \mathcal{T}$
for $i \in \{1, 2, ..., dims - 1\}$ **do**
  determine p_i in A
  compute QR decomposition of $p_i(A_i) = Q_i R_i$
  compute $A_{i+1} = Q_i^{-1} A_i Q_i + \kappa_i I$
end

Definition 15 (Single shifted QR-Algorithm for eigenvalue calculation)**.**

3.7 General functionality and wording

3.7.1 Abstract syntax tree

The abstract syntax tree [Noo85] (abbr. AST) is one of the most general and well-defined models in informatics. It describes the structure of a program as a tree. Every single functionality is packed in nodes where the edges illustrate the relationship between caller and parameter. Here is a small example:

$$val\, result\ : Real = 3 + (7 * 2) \tag{25}$$

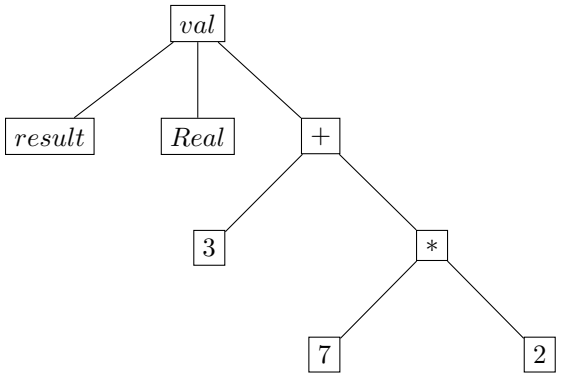

Figure 3: Example of a simple abstract syntax tree

The example shows how value has been declared with an alias or name, a data type and an initial value. The initial value is the result of a calculation. Already in this first step, the delimiters : and () were deleted because in an AST they are irrelevant. The lexical analyser is the first step in a parsing pipeline. It reads the code and setup tokens for each keyword. In my example, those keywords stand at the inner nodes of the tree, so *val* is a keyword for declaring a new value.

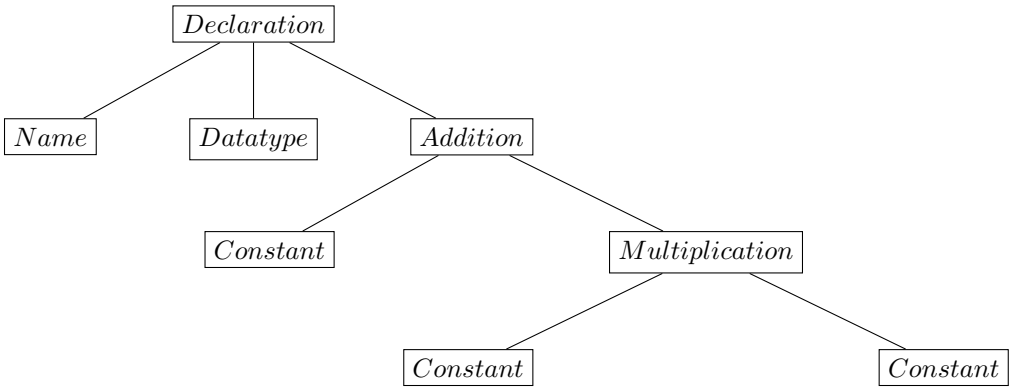

Figure 4: Example of a simple abstract syntax tree with denomination

In this section, all nodes are assigned to a denomination, which appropriates to the meaning of the node. This is the first step which is done in the parser.

At last, I want to show what is the difference between expressions [Noaa] and statements [Noad]. Expressions construct new values and return them without saving, while a statement is the order to execute an action. Whereat, in programming languages it is not always clear if its an expression or statement.

To illustrate this situation, there is a simple natural language example:

- Expression: "Tell me your position!"
- Statement: "Walk 1m forward!"
- Both: "Walk forward and then tell me your start position!"

In the many programming languages, it is possible to get return values of usual statements such as printing to check if the method has been accomplished without failures. In ExaStencils nodes are explicitly statements or expressions for the current layer (e.g. L4_Expression).

3.7.2 Steps in generation pipeline

The compilation pipeline includes the parser with the lexical analysis and the compiler coarsely, whereas code generation enlarges the workaround about a further step the generation. The next topics want to distinguish these steps.

Code generation does comparable steps as the compilation with two notable differences. The first is that the result is not a byte code but another source code language, either it is human-readable. Another difference is the input. On the one hand, compilation uses programming languages. On the other hand, code generation takes script languages, mostly. There are a few time steps to name, the generation time, the compilation time, and the runtime.

Generation time is the additional pipeline step of a code generator (e.g., ExaStencils), where the source code is transformed into the target code. As told, the result of the generation is human-readable, so the result is easy to control. Hence, a software developer can handle the generator without detailed assembler-knowledge, and white-box-testing can be arranged. At generation time, optimizations can be applied, which are only possible if a deeper understanding of the problem is existing, for instance, graph-minimization, apply special parallelization, or use the structure of the problem data.

Compilation time means the classical compilation, where the target code will be parsed, respectively analyzed, and byte-code will be outputted. In a typical use-case, software engineers develop target code. In the case of code generation, the target code will be produced by the generator. Tools for the compilation are, for example, gcc or clang. The compiler analyzes the target code for optimizations, e.g., loop-unrolling. At last, the compiler builds a runnable program.

The last step is the runtime, which means when the program is working on a processor. At this moment, only ad hoc optimizations can be applied, for example, branch-prediction.

Apparently, the most significant potential for optimization and failure prevention is at compilation time. Consequently, for the work with a code-generator, these capabilities have to be ensured.

3.7.3 PrettyPrintable

```
case class L4_Scope(var body : ListBuffer[L4_Statement]) extends L4_Statement {
  override def prettyprint(out : PpStream) : Unit = {
    out << "{\n"
    out <<< (body, "\n")
    out << "\n}"
  }

  override def progress = ProgressLocation(IR_Scope(body.map(_.progress)))
}
```

Listing 1: ExaStencils: Example prettyprintable node

If a node is pretty printable it derives from the PrettyPrintable trait and implements the method prettyprint(out). That allows to print every node to a predefined output, globally. The prettyprint method is used for controlling of parser results and for printing the generated code. Each layer is parsed twice. During the first time, the generator parses the handwritten ExaSlang code. After that, ExaStencils uses prettyprint to print the whole parsed AST again in ExaSlang (same layer). ExaStencils parses the printed ExaSlang code another time and checks if the second AST is similar to the first.

3.7.4 Progressable

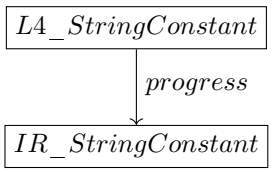

Figure 5: Example for a progressable node

```
case class L4_StringConstant(var value : String) extends L4_ConstantExpression {
  override def prettyprint(out : PpStream) : Unit = out << '"' << value << '"'
  override def progress = ProgressLocation(IR_StringConstant(value))
}
```

Listing 2: ExaStencils: Example progressable node

Preceding topics tells that ExaStencils has a layered approach and that each layer transforms the AST respectively the nodes to the next deeper layer. The usual way is the progress method, which is derived from the Progressable trait in each layer (e.g. L4_Progressable). If a node has a progress method, it is directly progressable to the next layer. Some nodes are progressable but will not be progressed this way. For these nodes, transformations are available which transfer the node to another before progress can be applied.

3.7.5 Transformation

$$print(\text{"Hello, "}); print(\text{"I am a string"}) \tag{26}$$
$$\widehat{=} print(\text{"Hello, "} + \text{"I am a string"}) \tag{27}$$

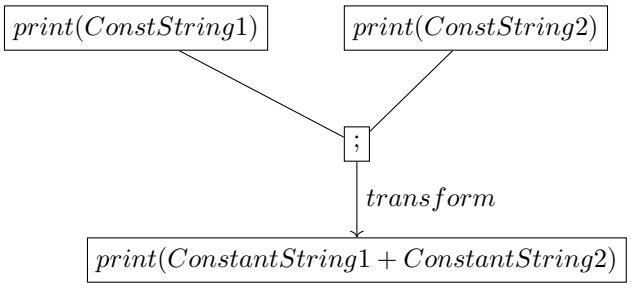

Figure 6: Example simple transformation

Generally, in languages, natural and formal, there are differences between what is written and what is meant. There are many possibilities to close this gap. In programming languages, there are transformations, which look on the AST, and depending on the found nodes and how they are ordered and connected, they transform these nodes to another. Transformations also have further responsibilities, for example, to simplify the AST or to apply the cheapest convenient method.

In the example, two print statements on constant strings were combined to one statement. Let us ask if this is the only possible transformation. Certainly not and neither it is consistently compliant to transform these statements. A common transformation for the example is adding whitespace e.g. line breaks, whereas for code printing semicolons could be added.

In ExaStencils, transformations are a part of a strategy, which means that each transformation is in the context of others. Collectively they define the interpretation of an ExaSlang script and how the resulting code is operating.

```
object PrintToFile extends DefaultStrategy(
    "Prettyprint all file-prettyprintable nodes"
  ) {
    this += new Transformation("Print", {
      case printable : FilePrettyPrintable =>
      printable.printToFile()
      printable
    })
}
```

Listing 3: ExaStencils: Example transformation

A transformation in ExaStencils is a Transformation-object which has two obligatory parameters, a name and a scala function which implements a match-operator that checks if the subtree matches to the transformation. The object is included in another object which extends the strategy.

3.7.6 Layer handler

A layer handler is a tool, which organizes one layer by applying for example parsing, transformations or at least progressing. These tools are located in the app module. Each layer has its handler, which is derived from LayerHandler[1]. It is calling the necessary steps peu á peu.

4 Examples for special tensors

This section gives a short insight into where tensors are used. As written before, tensors are broadly applied formalism to describe physical interrelations. The three examples are distinct in the kind of application, the modeled object's class, and the dimensionality. For this reason, they should help the reader of the thesis to acknowledge how helpful tensor can be for frameworks comparable to ExaStencils.

[1] location: exastencils/app/LayerHandler.scala

4.1 Cauchy stress tensor

Dissecting a domain Ω and two disjunct areas Ω_1, Ω_2, the Cauchy stress tensor σ describes the force on the surface Γ between the areas. This appears, for instance, if a part of the domain/the object Ω changes its temperature and thereupon, the distance between two grid points, maybe molecules, expands.

Definition 16 (Cauchy stress tensor)**.** *For* $[t] = [\sigma][n]$ *with*

$$[t] = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}, [\sigma] = \begin{bmatrix} \sigma_{11} & \sigma_{21} & \sigma_{31} \\ \sigma_{12} & \sigma_{22} & \sigma_{32} \\ \sigma_{13} & \sigma_{23} & \sigma_{33} \end{bmatrix}, [n] = \begin{bmatrix} n_1 \\ n_2 \\ n_3 \end{bmatrix}$$

$[\sigma]$ *is called the Cauchy stress tensor [Hol00].*

This thesis calls the exampled tensor the Cauchy stress tensor, but it is possible to find the name Cauchy stress matrix in other literature, which is a synonym.

4.2 Inertia tensor

The inertia tensor [Has06] (Θ, I) describes the inertia of a matter to change its angular momentum, therefore it is used in rigid body physics. May $\overrightarrow{L}$ the angular moment and $\overrightarrow{w}$ the angular velocity, then their relationship can be written as $\overrightarrow{L} = \Theta \cdot \overrightarrow{w}$.

Definition 17 (Inertia tensor)**.** *By looking at a rigid object with* N *point masses* m_k *and* $r_k = (x_1^{(k)}, x_2^{(k)}, x_3^{(k)})$ *the vector to the point mass, then the inertia tensor is*

$$\Theta = \begin{bmatrix} \Theta_{11} & \Theta_{12} & \Theta_{13} \\ \Theta_{21} & \Theta_{22} & \Theta_{23} \\ \Theta_{31} & \Theta_{32} & \Theta_{33} \end{bmatrix}$$

, with its componets

$$\Theta_{ij} = \sum_{k=1}^{N} m_k(||r_k||^2 \delta_{ij} - x_i^{(k)} x_j^{(k)})$$

.

4.3 Electromagnetic field tensor

The electromagnetic field tensor F_{uv} [San16] is used in theoretical physics and characterizes the electromagnetic field in Minkowski flat spacetime.

$$F_{uv} = \delta_u A_v - \delta_v A_u$$

describes the electromagnetism with a 4-dimensional vector potential $A^u = (\phi, A)$ and δ_u is the physics' four-gradient. F_{uv} represents the field strength, so the electromagnetic field tensor is also called field-strength tensor. F_{uv} is an antisymmetric tensor and has six independent components, where $E = (E_1, E_2, E_3) = (E_x, E_y, E_Z)$ is the electric field and $B = (B_1, B_2, B_3) = (B_x, B_y, B_z)$ is the magnetic field.

$$F_{uv} = \begin{bmatrix} 0 & -E_x & -E_y & -E_z \\ E_x & 0 & B_z & -B_y \\ E_y & -B_z & 0 & B_x \\ E_z & B_y & -B_x & 0 \end{bmatrix}$$

This thesis should not introduce this tensor in detail, so this image is concise for this complicated matter. For this work, it is interesting that there is a tensor that has its usability in electromagnetism/-dynamics. More precisely, physics uses it to simplify Maxwell's equation, which has a four-vector, to two tensor field equations. Therefore this instance shows a further scope of application where tensors are required.

5 Representation in ExaSlang

This section describes the transformation from a mathematical definition through the L4 to IR layer. I have chosen to illustrate the transformation in the natural working direction, whereas software development began on the last layer.

With the representation comes implicit the definition in the parser, therefore I am always giving this specification at hand.

May there be a second order tensor

$$second\_order\_tensor := \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 2.0 & 2.0 & 2.0 \\ 3.0 & 3.0 & 3.0 \end{bmatrix}$$

```
Var second_order_tensor : Tensor2< Real, 3 > = tens2{ 3,
    [0,0] => 1.0,
    [1,0] => 1.0,
    [2,0] => 1.0,
    [0,1] => 2.0,
    [1,1] => 2.0,
    [2,1] => 2.0,
    [0,2] => 3.0,
    [1,2] => 3.0,
    [2,2] => 3.0
}
```

Listing 4: ExaSlang4: Look-alike tensor representation

5.1 The tensor datatype

As described, a tensor [3.2] can have different orders and dimensions. These differences affect the functions. Therefore, some either are only defined for tensors with order one or two (e.g. determinant [14]) or the needs in the implementation are notably distinct (e.g. eigenvalues [15]). According to that, I added three datatypes for each tensor with order 1, 2 and higher, upon which these are derived from a collective type which will not be parsed. The datatype allocates the tensor array on the stack and defines the alias for the object.

```
Var t1 : TensorN< Real, 3, 4 > = initalValue
Var t2 : Tensor1< Real, 4 > = otherInitialValue
Var t3 : Tensor2< Real, 2 >
```

Listing 5: ExaSlang4: Example tensor datatypes

Language structure:

$$\mathbb{T} := TensorN < datatype, order, dimension > \tag{28}$$

$$t := Tensor1 < datatype, dimension > \tag{29}$$

$$\mathcal{T} := Tensor2 < datatype, dimension > \tag{30}$$

The first version of the tensor type only allows numerical data, e.g., real numbers. It is conceivable for a later version that complex data structures related to matrixes or tensors are allowed. $order, dimension \in \mathbb{N}$, so that the datatype allocates an array with $size = dimension^{order}$. For reducing memory abuse, order-one tensors should not be defined if it is not particularly necessary. Similarly, tensors with a high order or dimensionality should be reused instead of copied.

5.2 The tensor expression

In the most programming languages assignments has a LHS (left-hand-side) and a RHS (right-hand-side). The LHS is the variable with its type and the RHS is an expression of the same datatype. This is the reason why an extra expression is needed for a manually constructed tensor.

```
t1 = tensN{ 3; 4;
     [0,0,0,0] := 1.0,
     [1,0,0,0] := 1.0,
     [2,0,0,0] := 1.0,
     [0,1,0,0] := 2.0
}
t2 = tens1{ 3;
     [0] := 1.0,
     [1] := 1.0,
     [2] := 1.0
}
t2 = tens2{ 3;
     [0,0] := 1.0,
     [1,0] := 1.0,
     [2,0] := 1.0,
     [0,1] := 2.0
}
```

Listing 6: ExaSlang4: Example tensor expression

Language structure:

$$\mathbb{T} := tensN\{dimensionality; order; entries\} \tag{31}$$

$$t := tens1\{dimensionality; entries\} \tag{32}$$

$$\mathcal{T} := tensor2\{dimensionality; entries\} \tag{33}$$

Language structure entry:

$$te_{inner} := [index1, ...] := value, \tag{34}$$

$$te_{last} := [index1, ...] := value \tag{35}$$

Tensor expressions are logically divided into two parts. The outlines define the properties. On the other side, an entry holds the value on a specific coordinate. Tensors are sparse defined, which implies that the generator fills not determined entries with zero values. Each coordinate is as long as the given order. Notable is that the separators are altered in the first tensor implementation (”,” → ”;” and ” => ” → ” := ”).

5.3 Element access

The element access should be similar to other modern programming languages. An n-dimensional tensor has to implement a [ ]-Operator for each dimension.

```
Var t1 : Tensor2< Real, 3 > = initialValue

Val result : Real = 0.0
Var cnt : Int = 0
Var cnt2 : Int = 0

repeat 3 times count cnt {
    repeat 3 times count cnt2 {
        result += t1[cnt, cnt2]
    }
}
Val result_2 : Real = t1[2,2]
```

Listing 7: ExaSlang4: Example element access

Language structure:

$$\mathcal{T}_{ij} := (Tensor1 < datatype >)[index1, index2] \tag{36}$$

In the example and the language structure, only the 2-dimensional access is represented, naturally, if the tensor has an order greater than 2 there have to be more indices.

5.4 Arithmetic operations

The arithmetic operators are attempted to be in a natural way, which means that we only use the classic mathematically operators $+, -, *, /$.

```
Var t1 : Tensor2< Real, 3 > = initialValue1
Var t2 : Tensor2< Real, 3 > = initialValue2
Var scl : Real = 2.3

Val result : Tensor2< Real, 3 > = t1 + t2
Val result_2 : Tensor2< Real, 3 > = t1 * scl
Val result_3 : Tensor2< Real, 3 > = scl / t2
```

Listing 8: ExaSlang4: Example arithmetic operations

I do not show an example of all possible operators here. The implemented operators are described in the theory section before [3.4].

5.5 Other functions

In mathematics the infix-notation is most common for functions. So it is reasonable that infix is also chosen here.

```
Var t1 : Tensor2< Real, 3 > = initialValue
Var t2 : TensorN< Real, 4, 1 > = initalValue2
Var t3 : TensorN< Real, 5, 2 > = initialValue3

Val determinant : Real = deter(t1)
Val trace : Real = trace(t1)

Val vec : Matrix< Real, 3, 1 >
eigen(t1, vec)

Val tensor1_result = asTensor1(t2)
Val tensor2_result = asTensor2(t3)
```

Listing 9: ExaSlang4: Example function call

The functions deter [14] and trace [12] return their results, instead of eigen [15] were the result vector has been passed as a parameter. The eigenvalue method is the only function that works on runtime, all other functions were calculated on generation-time.

5.6 Dyadic product

In most sources, the dyadic product [1] is written with the $*$, or $\times$ operator. We choose an infix notation as the other functions to avoid collisions with the arithmetic operators.

```
Var m1 : Matrix< Real, 3, 1 > = initialValue
Var m2 : Matrix< Real, 3, 1 > = initialValue2
Var t1 : Tensor1< Real, 3 > = initialValue3
Var t2 : Tensor1< Real, 3 > = initialValue4

Var result_tensor2 : Tensor2< Real, 3 > = dyadic(m1, m2)
Var result_tensor3 : TensorN< Real, 3, 3 > =
    dyadic(result_tensor2, t1)
Var result_tensor4 : TensorN< Real, 3, 4 > =
    dyadic(result_tensor3, t1)

Var result_tensor2_2 : Tensor2< Real, 3 > = dyadic(t1, t2)
Var result_tensor4_2 : TensorN< Real, 3, 4 > =
    dyadic(result_tensor2, result_tensor2_2)
```

Listing 10: ExaSlang4: Example dyadic product

Order 2 tensors can be constructed out of matrix and order 1 tensors by calculating the dyadic product. Comparatively, tensors with a higher order can also be constructed that way, although only out of tensors. The implementation do not differ between dyadic and tensor product.

5.7 Einstein notation

As in section Einstein notation [2] is described. It is an abbreviation for the aggregation over iteration variables.

```
Var t1 : Tensor2< Real, 3 > = initialValue
Var t2 : Tensor2< Real, 3 > = initialValue2
Var t3 : Tensor2< Real, 3 > = initialValue3

Val result : Real = t1[a, 0] * t2[a, 1]
Val result_2 : Real = t1[2, a] / t2[1, a]
Val result_3 : Real =
    t1[a, 1] + t3[a, 2] * t2[2, b] * t2[0, b] / t3[c,c]
Val result_4 : Real = t1[2, a] / t2[1, a] - t3[0, a]
```

Listing 11: ExaSlang4: Example Einstein notation

This example calculates:

$$result = \sum_{k=0}^{2} t1_{k,0} * t2_{k,1} \tag{37}$$

$$result_2 = \sum_{k=0}^{2} t1_{2,k} / t2_{1,k} \tag{38}$$

$$result_3 = \frac{\left(\sum_{k=0}^{2} t1_{k,1} + t3_{k,2}\right) * \left(\sum_{k=0}^{2} t2_{2,k} * t2_{0,k}\right)}{\sum_{k=0}^{8} t3_k} \tag{39}$$

$$result_4 = \sum_{k=0}^{2} t1_{2,k} / t2_{1,k} - t3_{0,k} \tag{40}$$

Language structure:

$$tensor1[index1, index2] \circ tensor2[index3, index4] \tag{41}$$

An explicit parser definition is not necessary because the expression will be parsed as it is, a binary operator ($\circ$) with two-element access ($tensor1[index1, index2]$ and $tensor2[index3, index4]$) where minimum two indices are doubled and Einstein iterators. The transformation to an Einstein notation will be done in IR-layer, which is described later in the implementation. A crucial characteristic of an Einstein notation is the free indices in the bracket access. We call such free indices Einstein iterators. The repeated iterator will be used for aggregation.

Definition 18 (Einstein iterator)**.** *May ϕ a locally fresh variable [GP02], which means that name is unused and not filled with any value, then we call ϕ as an Einstein iterator if it is contained by a bracket element access.*

When looking at the $result_3$ statement and the mathematical expression below, it can be recognized that the Einstein notation will be applied before the operator was bounded. Usually, $*$ comes before $+$, but in this example, the operator is bounded firstly on the existing similar einstein iterator a. Consequently, $t1[a, 1] + t3[a, 2]$ bounds more than $t3[a, 2] * t2[2, b]$. Otherwise, in a notation, multiplication/division comes first.

Definition 19 (Rules for Einstein notations in ExaSlang)**.** *May i an einstein iterator, a, b, c (non-essential different) constants. $\circ, \bullet$ are arbitrary binary operators were $\circ$ bounds more than $\bullet$. May also $\mathcal{k}, \ell, m, \mathcal{k}^T, \ell^T, m^T$ first-order tensors, $\mathcal{T}, \mathcal{U}, \mathcal{V}, \mathcal{W}$ (non-essential different) second-order tensors, whereat all tensors have the same dimensionaly. May be $(.., ..)$ a transformable Einstein notation and all indexes are similar to bracket operators. Then applies:*

Resolution rules :

$$(\mathcal{T}_{i,a} \circ \mathcal{U}_{i,b}) \to \mathcal{k}_i$$
$$(\mathcal{T}_{a,i} \circ \mathcal{U}_{b,i}) \to \mathcal{k}_i^T$$
$$(\mathcal{k}_i \circ \mathcal{T}_{i,b}) \to \ell_i$$
$$(\mathcal{T}_{i,a} \circ \mathcal{k}_i) \to \ell_i$$
$$(\mathcal{k}_i^T \circ \mathcal{T}_{b,i}) \to \ell_i^T$$
$$(t_{b,i} \circ \mathcal{k}_i^T) \to \ell_i^T$$
$$(\mathcal{k}_i \circ \ell_i) \to m_i$$
$$(\mathcal{k}_i^T \circ \ell_i^T) \to m_i^T$$

Transformation rules :

$$\mathcal{T}_{i,a} \circ \mathcal{U}_{i,b} \to (\mathcal{T}_{i,a} \circ \mathcal{U}_{i,b})$$
$$\mathcal{T}_{a,i} \circ \mathcal{U}_{b,i} \to (\mathcal{T}_{a,i} \circ \mathcal{U}_{b,i})$$
$$\mathcal{k}_i \circ \ell_i \to (\mathcal{k}_i \circ \ell_i)$$
$$\mathcal{k}_i^T \circ \ell_i^T \to (\mathcal{k}_i^T \circ \ell_i^T)$$
$$\mathcal{T}_{i,a} \circ \mathcal{k}_i \to (\mathcal{T}_{i,a} \circ \mathcal{k}_i)$$
$$\mathcal{k}_i \circ \mathcal{T}_{i,a} \to (\mathcal{k}_i \circ \mathcal{T}_{i,a})$$
$$\mathcal{T}_{a,i} \circ \mathcal{k}_i^T \to (\mathcal{T}_{a,i} \circ \mathcal{k}_i^T)$$
$$\mathcal{k}_i^T \circ \mathcal{T}_{a,i} \to (\mathcal{k}_i^T \circ \mathcal{T}_{a,i})$$

Preference rules :

$$\mathcal{T}_{i,a} \circ \mathcal{U}_{i,b} \bullet \mathcal{V}_{i,c} \to (\mathcal{T}_{i,a} \circ \mathcal{U}_{i,b}) \bullet \mathcal{V}_{i,c}$$
$$\mathcal{T}_{i,a} \bullet \mathcal{U}_{i,b} \circ \mathcal{V}_{i,c} \to \mathcal{T}_{i,a} \bullet (\mathcal{U}_{i,b} \circ \mathcal{V}_{i,c})$$
$$\mathcal{T}_{i,a} \circ \mathcal{U}_{i,b} \circ \mathcal{V}_{i,c} \to (\mathcal{T}_{i,a} \circ \mathcal{U}_{i,b}) \circ \mathcal{V}_{i,c}$$
$$\mathcal{T}_{a,i} \circ \mathcal{U}_{b,i} \bullet \mathcal{V}_{a,i} \to (\mathcal{T}_{a,i} \circ \mathcal{U}_{b,i}) \bullet \mathcal{V}_{c,i}$$
$$\mathcal{T}_{a,i} \bullet \mathcal{U}_{b,i} \circ \mathcal{V}_{c,i} \to \mathcal{T}_{a,i} \bullet (\mathcal{U}_{b,i} \circ \mathcal{V}_{c,i})$$
$$\mathcal{T}_{a,i} \circ \mathcal{U}_{b,i} \circ \mathcal{V}_{c,i} \to (\mathcal{T}_{a,i} \circ \mathcal{U}_{b,i}) \circ \mathcal{V}_{c,i}$$
$$\mathcal{k}_i \circ \ell_i \bullet m_i \to (\mathcal{k}_i \circ \ell_i) \bullet m_i$$
$$\mathcal{k}_i \bullet \ell_i \circ m_i \to \mathcal{k}_i \bullet (\ell_i \circ m_i)$$
$$\mathcal{k}_i \circ \ell_i \circ m_i \to (\mathcal{k}_i \circ \ell_i) \circ m_i$$
$$\mathcal{k}_i^T \circ \ell_i^T \bullet m_i^T \to (\mathcal{k}_i^T \circ \ell_i^T) \bullet m_i^T$$
$$\mathcal{k}_i^T \bullet \ell_i^T \circ m_i^T \to \mathcal{k}_i^T \bullet (\ell_i^T \circ m_i^T)$$
$$\mathcal{k}_i^T \circ \ell_i^T \circ m_i^T \to (\mathcal{k}_i^T \circ \ell_i^T) \circ m_i^T$$
$$\mathcal{T}_{i,a} \circ \mathcal{k}_i \bullet \mathcal{V}_{i,c} \to (\mathcal{T}_{i,a} \circ \mathcal{k}_i) \bullet \mathcal{V}_{i,c}$$
$$\mathcal{T}_{i,a} \bullet \mathcal{k}_i \circ \mathcal{V}_{i,c} \to \mathcal{T}_{i,a} \bullet (\mathcal{k}_i \circ \mathcal{V}_{i,c})$$
$$\mathcal{T}_{i,a} \circ \mathcal{k}_i \circ \mathcal{V}_{i,c} \to (\mathcal{T}_{i,a} \circ \mathcal{k}_i) \circ \mathcal{V}_{i,c}$$
$$\mathcal{T}_{a,i} \circ \mathcal{k}_i^T \bullet \mathcal{V}_{a,i} \to (\mathcal{T}_{a,i} \circ \mathcal{k}_i^T) \bullet \mathcal{V}_{c,i}$$
$$\mathcal{T}_{a,i} \bullet \mathcal{k}_i^T \circ \mathcal{V}_{c,i} \to \mathcal{T}_{a,i} \bullet (\mathcal{k}_i^T \circ \mathcal{V}_{c,i})$$
$$\mathcal{T}_{a,i} \circ \mathcal{k}_i^T \circ \mathcal{V}_{c,i} \to (\mathcal{T}_{a,i} \circ \mathcal{k}_i^T) \circ \mathcal{V}_{c,i}$$

The rules are defined so that if more than one rule matchs, the rule with the most operators applies first. Each rule can stand in the context of another term.

Writing the calculation-rules related to term rewriting systems [Klo00] is not the most understandable presentation, but it discloses the Einstein notation module's size. In the first version of this module, the transformation can only apply to tenors with the order two or one, excluding

double contraction, which is described below. There are some possible meaningful extensions for the future, which were not implementable at the time of the thesis. For instance, another iteration variables could be applied, or instead of the summation at the end, the user could save the resulted vector from the last transformation. Another interesting topic could be the universal rule definition of an Einstein notation from an informatic view so that it is possible to use any tensors in the module.

5.8 Double contraction

The double contraction [10] is a part of the Einstein notation, so the state has to be noted with indices and an operator in the middle. I avoid the classic mathematical notation $\mathcal{T} : \mathcal{U}$ or $\mathbb{T} : \mathcal{U}$ because now the : operator can be used for elementwise division and it is possible to choose the operator in the contraction freely.

```
Var t1 : Tensor2< Real, 3 > = initialValue1
Var t2 : Tensor2< Real, 3 > = initialValue2
Var t3 : TensorN< Real, 3, 4 > = initialValue3

Val result : Real = t1[a,b] : t2[a,b]
Val result_2 : Tensor2< Real, 2 > = t3[a,b,c,d] * t1[c,d]
Val L2Norm : Real = sqrt( t1[a,b] * t1[a, b] )
```

Listing 12: ExaSlang4: Example double conraction

To apply double contraction, the last two indices have to be doubled as in the example. Moreover, for order four to order two contractions, the first indices also have to be Einstein iterators. Notable is that the order of the factors is the same as in most sources for tensor calculation. Therefore the order four tensor has to be written before the order two tensor.

The last result $L2Norm$ is a simple example of how Einstein notation and the double contraction can build other functions, which actually calculates $L2Norm = ||t1||_2 = (\sum_i t1_i^2)^{\frac{1}{2}}$

6 Implementation in ExaStencils

6.1 Coorperative work

Fabian Böhm defines and implements in his bachelor thesis [Boe20] the matrix data type. The matrix have a related functionality and lookalike as first- and second-order tensors. Mr Böhm and I implemented the same way for linearization, or more generally storage type, to make tensors and matrix usable in both classes. To enable a smooth collaboration, we work together to synchronize our code states in frequent merges. Also, we include guards to function calls to avoid incorrect usage.

The matrix is used to construct tensors and together with tensors in the arithmetic part. The access module and the Einstein notation are explicitly implemented for both data types.

The cooperation only includes the harmonization of the data types and that they can work side by side. Mr. Boehm and I do not implement in others' work. Nevertheless, we help each other with troubleshooting.

Both datatypes, the matrix, and the tensor type can work together. This thesis describes the functionality, which is fully implemented with tensors and works autonomously from a matrix. Exclusively, at the einstein notation module, a matrix in the form of a transformed vector is used to differ in which order was contracted.

```
Var m1 : Matrix< Real, 3 , 3> = initialValue
Var t1 : Tensor2< Real, 3 > = asTensor2(m1)

Var m2 : Matrix< Real, 3 , 3> = initialValue2
Var t2 : Tensor1< Real, 3 > = asTensor1(m2)
```

Listing 13: ExaSlang4: Example conversion

Module	Desription
Constructors	Uses matrix as an input type for dyadic product.
Arithmetic	Uses matrix for all classical arithmetic operators $(+, -, *, /)$.
Access	The classic bracket operator works also with matrix. matrix are used in the complex access module.
EinsteinNotation	With the Einstein notation matrix can be used instead of tensors.
HigherFunctions	matrix can be convertet to first- and second order tensors

Table 1: Table of cooperative modules.

The listing [13] shows how a matrix can be transformed into a tensor. In this version, ExaStencils only transforms $n \times n$ matrixes into order-two tensors and vector-related matrixes into order-one tensors. At a later version, it could be reasonable to convert higher orders, which means that matrixes with matrixes as an internal datatype must exist. The asTensor1 and asTensor2 function controls if the inputted matrix has a compatible form and datatype. Tensors with order one do not have a direction as vector-alike matrixes (row-vector or column-vector), so the function ignores this specification.

6.2 Modules

These are the lists of changed or created L4 and IR modules to apply the tensor datatype. The L4 modules are relatively short, whereas the IR modules are longer and much more complicated. All these modules are submodules of Compiler/src/.

Module	Desription
L4_Parser	Defintion of the layer 4 parser
L4_Lexer	Definition of the key words and delimiters
L4_LayerHandler	Applies the L4 parser and transform the parsed statements to the IR layer
L4_HigherDimensionalDatatypes	Definition of the tensor datatype in layer 4
L4_TensorAccess	Definition of the tensor expression in layer 4 with its constructors
L4_ComplexAccess	Resolves the multidimensional and more complex access to tensors and matrixes

Table 2: Table of changed L4 modules.

6.3 Layer 4

6.3.1 About the parser

ExaStencils follows the usual structure where a lexical analyser (abbr. lexer) destructs the input string (source code) to smaller substrings by setting up tokens for each found keyword. In the next step, a parser consumes each token and its substring and handles them basing on predefined rules.

In ExaStencils the lexer called (ExaLexer [2]) has a distinct definition for each layer excluding IR, where keywords and delimiters are specified. To define tensors on the *L4_ Lexer* [3], keywords for both datatypes (*Tensor*, *Tensor1*, *Tensor2*, *TensorN*) and expressions (*tens1*, *tens2*, *tensN*) were added to the definiton. The parser in ExaStencils (ExaParser [4]) is a so called packrat parser [For],

[2] location: exastencils/parsers/ExaLexer.scala
[3] location: exastencils/parsers/l4/L4_Lexer.scala
[4] location: exastencils/parsers/ExaParser.scala

Module	Desription
IR_LayerHandler	Applies the transformations on the AST and prints out the resulting c++ code.
IR_HigherDimensionalDatatypes	Definition of the tensor datatype in the IR layer
IR_TensorAccess	Definition of the tensor expression in the IR layer with its constructors
IR_ResultingDatatype	Determines the resulting datatype
IR_TensorAssingments	Resolves the assignments of tensor accesses and expressions
IR_ResolveTensorReturnValues	Defines the names of the tensor member functions and implements their return
IR_ResolveTensorCompiletimeFunctions	Defines, implements and resolves the tensor member functions, which has been calculated to compile time (e.g. multiplication)
IR_ResolveTensoreRuntimeFunctions	Defines, implements and resolves the tensor member functions, which has been calculated to runtime (e.g. eigenvalures)
IR_ComplexAccess	Implements and resolves the multidimensional and more complex access to tensors and matrix. It also evaluates the return of an Einstein notations
IR_ResolveEinsteinNotation	Resolves the Einstein notation, by mapping the similar char-like indices of a complex access

Table 3: Table of changed IR modules.

which is a top-down type. Those parsers handle at first the most left token. The ExaParser steps down recursively into each substring and treat it as the entire source code.

6.3.2 Tensor datatype

6.3.2.1 Parser definition

```
lazy val algorithmicDatatype : Parser[L4_Datatype] = (
  "TensorN" ~ ("<" ~> numericDatatype <~ ",") ~ (integerLit <~ ",") ~
    (integerLit <~ ">") ^^
    { case _ ~ x ~ dims ~ order => L4_TensorDatatypeN(x, dims, order) }
  ||| "Tensor1" ~ ("<" ~> numericDatatype <~ ",") ~ (integerLit <~ ">") ^^
    { case _ ~ x ~ dims => L4_TensorDatatype1(x, dims) }
  ||| "Tensor2" ~ ("<" ~> numericDatatype <~ ",") ~ (integerLit <~ ">") ^^
    { case _ ~ x ~ dims => L4_TensorDatatype2(x, dims) }
)
```

Listing 14: L4_Parser: Defintion of tensor datatypes

6.3.2.2 Class definition

The tensor datatype object[5] is located at the baseExt folder and derives from the *L4_HigherDimensionalDatatype* which contains only complex datatypes e.g. matrix.

```
abstract class L4_TensorDatatype(datatype : L4_Datatype, dims : Int)
  extends L4_HigherDimensionalDatatype {
    override def prettyprint(out : PpStream)
    override def progress: IR_HigherDimensionalDatatype
    // ...
}
```

Listing 15: ExaStencils: Layer 4, Tensor datatype

[5]location: exastencils/baseExt/l4/L4_HigherDimensionalDatatype.scala

6.3.2.3 Description

As previously explained, there are three types of tensors and an abstract tensor where the others have derived from. The member functions of the tensor overwrite the functions from the *L4_Datatype* class, where all data types are derived. The data type has obligatorily two parameters *datatype*, which determines the inner type, and *dims* that held the dimensionality of the tensor. For the n-th-order tensor, a third parameter is relevant, the so called *order* parameter which indicates the order of the tensor. The listing [14] shows the parser definition for the tensor datatype. *TensorN*, *Tensor1*, and *Tensor2* are the keywords for the tensor. These are defined at the *L4_Lexer*. " <>, () are delimiters, which also noted at the lexer definition.

6.3.3 Tensor expression

6.3.3.1 Parser definition

```
lazy val tensorExpression1 =
  locationize(
    ("tens1" ~ "{") ~> (integerLit <~ ";") ~ (repsep(tensorEntry, ",") <~ "}") ^^
    {case dims ~ x => L4_TensorExpression1(None, dims, x) }
  )

lazy val tensorExpression2 =
  locationize(
    ("tens2" ~ "{") ~> (integerLit <~ ";") ~ (repsep(tensorEntry, ",") <~ "}") ^^
    {case dims ~ x => L4_TensorExpression2(None, dims, x) }
  )

lazy val tensorExpressionN =
    locationize(
      ("tensN" ~ "{") ~> (integerLit <~ ";") ~ (integerLit <~ ";") ~
      (repsep(tensorEntry, ",") <~ "}") ^^
      {case dims ~ order ~ x => L4_TensorExpressionN(None, dims, order, x) }
    )

lazy val tensorEntry =
    locationize(("[" ~> repsep(integerLit, ",") <~ "]") ~
    (":=") ~> numLit) ^^
    { case index ~ coeff => L4_TensorEntry(index, coeff) })
```

Listing 16: L4_Parser: Defintion of tensor expression

6.3.3.2 Class definition

```
case class L4_TensorEntry(var index : List[Int], var coefficient : L4_Expression)
  extends L4_Node with L4_Progressable with PrettyPrintable {
  override def prettyprint(out : PpStream) = out << "[" << index.mkString(",")
    << "]" << ":=" << coefficient
  override def progress = null

  def convertConstants(dt : L4_Datatype) : Unit = (coefficient, dt) match {
    // ...
  }
}

case class L4_TensorExpression1(
  var datatype : Option[L4_Datatype],
  var dims : Integer,
  var expressions : List[L4_TensorEntry]) extends L4_Expression {

  def prettyprint(out : PpStream) = {
    out << "tens1" << "{" << dims.toString << ";" <<< (expressions, ", ") << "}"
 }

  override def progress = ProgressLocation(IR_TensorExpression1(
    L4_ProgressOption(datatype)(_.progress),
    dims,
    progressEntrys(dims, expressions)
))

  def progressEntrys(dims: Int, input : List[L4_TensorEntry]) :
    Array[IR_Expression] = {
    // ...
  }

  def order = 1
  def isConstant = expressions.count(_.isInstanceOf[L4_Number]) ==
    expressions.length
  def convertConstants(dt : L4_Datatype) : Unit = {
    expressions.foreach(_.convertConstants(dt))
  }
}
```

Listing 17: ExaStencils: Layer 4, Tensor expression

6.3.3.3 Description

Comparable to the tensor data type for the expression, three types are also implemented at the fourth layer. On the other hand, the tensor expression derives from *L4_Expression* directly and not over an own collective class. The elements of a tensor have an own class on the fourth layer, the *L4_ TensorEntry* holds the coordinate from ExaSlang and its value as well. At the progress step, tensor expressions and its entries will be transformed. The listing of the tensor expression [17] should outline a few details. The first topic is that two object types necessary for an expression. The tensor entry is a container of a single entry and the expression, which contains and progresses the entries. This structure is far from the IR-layer, which does not have its entry type. Because of this reason, the tensor entry does not share a progress method. The progressing is a part of the tensor expression. It has to combine entry and coordinate and sort it into an array. At this point, the empty coordinates will be estimated and set to zero value.

6.3.4 Element access

6.3.4.1 Parser definition

```
lazy val genericAccess = (
  locationize(ident ~ slotAccess.? ~ levelAccess.? ~ ("@" ~> constIndex).? ~
    ("[" ~> integerLit <~ "]").? ~ ("[" ~> complexMulDimIndex <~ "]")
  ^^ { case id ~ slot ~ level ~ offset ~ arrayIndex ~ mulDimIndex =>
    L4_UnresolvedAccess(id, level, slot, offset, None, arrayIndex, mulDimIndex) })

  // ...
)

lazy val complexMulDimIndex = (
  (flatAccess ||| numLit) ~ ("," ~> (flatAccess ||| numLit)).+
    ^^ { case entry ~ entries => (entry) :: entries }
)
```

Listing 18: L4_Parser: Definition of element Access

6.3.4.2 Class definition

```
case class L4_UnresolvedAccess(
  var name : String,
  var level : Option[L4_AccessLevelSpecification],
  // ...
  var arrayIndex : Option[Int],
  var mulDimIndex : List[L4_Expression])
    extends L4_Access with L4_CanBeOffset {
  // ...
}

case class L4_ComplexAccess(
  var name : String,
  var level : Option[L4_AccessLevelSpecification],
  var arrayIndex : Option[String],
  var mulDimIndex : List[L4_Expression]) extends L4_Access {

  var decl : L4_VariableDeclaration = null

  def setDecl(x : L4_VariableDeclaration) = {
    decl = Duplicate(x)
  }

  // ...
}
```

Listing 19: ExaStencils: Layer 4, Element/Complex Access

6.3.4.3 Transformation

```
object L4_ResolveComplexAccess
  extends DefaultStrategy("Resolve generation of L4 Complex Access") {

  this += new Transformation("Transform an unresolved to a complex access", {
    case access : L4_UnresolvedAccess if (
      access.asInstanceOf[L4_UnresolvedAccess].mulDimIndex.nonEmpty
    ) =>
      L4_ComplexAccess(access.name, access.level, None, access.mulDimIndex)
    })
}


object L4_ValidateComplexAccess
  extends DefaultStrategy("Resolve validation of L4 Complex Access") {
  // ...

  def myresolve(access : L4_ComplexAccess) : L4_ComplexAccess = {
    access.setDecl(declCollector.getDeclaration(access.name))
    access
  }

  this += new Transformation("find assigned declaration and add to knot", {
    case access : L4_ComplexAccess => myresolve(access)
  })
}
```

Listing 20: ExaStencils: Layer 4, Transformations Complex Access

6.3.4.4 Description

The element access in the fourth layer and later on the IR layer is one of the most complex elements of my work. It is essential for the classical element access, as well as for more intricate moduls access method, such as Einstein notation [2] and contraction [10].

The first component is the parser definition, which builds a *L4_ UnresolvedAccess* with a complex multidimensional index. One index dimension can be a *flatAccess* which is an alias of a variable or an Einstein iterator, or on the other hand a *numLit*, that contains an integer to a single excerpt.

The second component is the definition of a class the *L4_ ComplexAccess* and enlarges the unresolved access with the complex index. Excluding *PrettyPrint* and *progress* the complex access on the fourth layer has no own functionality, it rather acts as an intermediate step to the IR layer.

As the last component of the element access on the fourth layer, two transformations have to be named. The *L4_ ResolveComplexAccess* transforms unresolved access to complex access if the origin node holds a multidimensional index, a name, and optionally, a level. The second transformation *L4_ ValidateComplexAcces* determines if the name parameter accommodates a legal name of a *L4_ VariableDeclaration*. Assuming that, the decl parameter gets the detected declaration.

6.4 IR Layer

6.4.1 Tensor datatype

6.4.1.1 Class definition

```
abstract class IR_TensorDatatype(datatype : IR_Datatype, dims : Int)
  extends IR_HigherDimensionalDatatype with IR_HasTypeAlias {

  override def prettyprint(out : PpStream)
  override def prettyprint_mpi = s"INVALID DATATYPE: "
    + this.prettyprint()

  override def dimensionality: Int
  override def getSizeArray: Array[Int]
  override def resolveDeclType: IR_Datatype
  override def resolveDeclPostscript : String = ""
  override def resolveFlattendSize: Int
  override def typicalByteSize: Int

  override def aliasFor = datatype.prettyprint
    + '[' + this.resolveFlattendSize + ']'
  def getDims : Int = dims
}
```

Listing 21: ExaStencils: IR Layer, Tensor datatype

6.4.1.2 Description

Similar to the fourth layer and also ExaSlang, the tensor datatype has three peculiarities in the IR layer. Differing from the upper layers, the *PrettyPrint* function prints now the node to the destination code. From the *IR_HasTypeAlias* class the *aliasFor* parameter is derived. Alias parameters will be printed to the header file and replaced with by a datatype in the target code. For example, a second-order tensor which holds *IR_RealDataype* as the inner datatype will be replaced by the alias data type *double*[9]. Because of the linearization of multidimensional datatypes, the array has the size of the flatten tensor.

6.4.2 Tensor expression

6.4.2.1 Class definition

```
abstract class IR_TensorExpression(
    innerDatatype : Option[IR_Datatype], dims : Integer
  ) extends IR_Expression {
  var expressions : Array[IR_Expression]
  val order : Integer

  override def datatype: IR_Datatype

  def prettyprintInner(out : PpStream) : Unit
  override def prettyprint(out : PpStream) : Unit

  def isConstant : Boolean = expressions.forall(e => e.isInstanceOf[IR_Number])
  def isInteger : Boolean =
    expressions.forall(e => e.isInstanceOf[IR_IntegerConstant])
  def isReal : Boolean = expressions.forall(e => e.isInstanceOf[IR_RealConstant])
  def toString : String
}
```

Listing 22: ExaStencils: IR Layer, Tensor expression

6.4.2.2 Description

The tensor expression in the IR layer is not so complicated, respectively it is the result of the progress step from the fourth layer. For the practical use of tensors in the IR layer are the functions *isInteger* and *isReal* crucial. With them it is possible to differ between a test on 'is similar' or on 'is similar in an epsilon environment'.

6.4.3 Element access

6.4.3.1 Class definition

```
case class IR_ComplexAccess(
    var name : String,
    var decl : IR_VariableDeclaration,
    var arrayIndex : Option[String],
    var mulDimIndex : List[IR_Expression],
    deep : Int
  ) extends IR_Access {

  override def prettyprint(out : PpStream) : Unit =
    out << "\n --- NOT VALID ; NODE_TYPE = " <<
  this.getClass.getName << "\n"
  override def datatype : IR_Datatype = decl.datatype

  def resolve() : IR_Expression = {
    // ...
  }
```

Listing 23: ExaStencils: IR Layer, Element/Complex Access

6.4.3.2 Transformation

```
object IR_ResolveComplexAccess
  extends DefaultStrategy("Resolve transormation of Complex Access") {

  def resolve(access : IR_ComplexAccess) : IR_Expression = {
    access.resolve()
  }
  this += new Transformation(
    "add assignments/decl to function returns to arguments", {
      case access : IR_ComplexAccess => resolve(access)
  })
}
```

Listing 24: ExaStencils: IR Layer, Transformations Complex Access

6.4.3.3 Description

The IR layer's complex-access will separate into three distinct parts by a match-case operator during its transformation. The primary transformation will be applied at the *access.resolve()*-method, which is described here.

Firstly, a classical access with constants (*IR_IntegerConstant*) is implemented, so it is possible to access to a single coordinate, e.g. $t1[0,2] \Rightarrow$ "$t1[0+2*3]$" $\Rightarrow IR\_HighDimAccess($ $acc, IR\_ExpressionIndex(IR\_IntegerConstant(x + y * tmp.dims))), x := 0 \wedge y := 2$. Constant coordinates are calculated at compilation time.

As a second part, open access is evolved. Here variable accesses and constant coordinates are possible, which allows that the module set up strings (*IR_StringLiteral*), determining the location of the requested element to runtime, e.g. $t1[0,i] \Rightarrow$ "$t1[0+i*3]$" $\Rightarrow IR\_StringLiteral($ $Array(name, "[", x, "+", y, "*", tmp.dims, "]").mkString("")), x := 0 \wedge y := i$. Generally, it is not intended to implement access as *IR_StringLiteral*. This tactic is used for free iteration variables (here i) while accessing linearized arrays directly.

Finally, if only a single Einstein iterator was found at an index, the transformation would summarize overall objects at this dimension. This part is only a return state of the Einstein notation, hence in detail, it will be delineated there.

At this state, an ArrayIndex exists at the parser definition and the class of the *L4_ComplexAccess*, but it is unused. The value is needed at the next version, where tensors and matrixes with one axis can be added to Einstein notation.

6.4.4 Einstein notation

6.4.4.1 Transformation

```
1  object IR_ResolveEinsteinNotation
2    extends DefaultStrategy("Resolve Einstein Notation") {
3
4    def resolve(
5      access1 : IR_ComplexAccess,
6      access2 : IR_ComplexAccess,
7      operator : Char
8    ) : IR_Expression = {
9
10     // ...
11
12 }
13
14 this += new Transformation("Apply Binary Operator for Einstein Notation", {
15 case call : IR_Multiplication if
16 ((call.asInstanceOf[IR_Multiplication].factors(0).isInstanceOf[IR_ComplexAccess]) &&
17 (call.asInstanceOf[IR_Multiplication].factors(1).isInstanceOf[IR_ComplexAccess])) =>
18   val tmp = call.asInstanceOf[IR_Multiplication]
19   if (tmp.factors.length > 2) {
20     var mul_list = myresolve(
21       tmp.factors(0).asInstanceOf[IR_ComplexAccess],
22       tmp.factors(1).asInstanceOf[IR_ComplexAccess],
23       '*'
24     )
25     for (i <- 2 until tmp.factors.length)
26       mul_list = resolve(
27         mul_list.asInstanceOf[IR_ComplexAccess],
28         tmp.factors(i).asInstanceOf[IR_ComplexAccess],
29         '*'
30       )
31     mul_list
32   } else {
33     resolve(tmp.factors(0).asInstanceOf[IR_ComplexAccess],
34     tmp.factors(1).asInstanceOf[IR_ComplexAccess], '*')
35   }
36
37   // ...
38
39   })
40 }
```

Listing 25: ExaStencils: IR Layer, Transformations Einstein notation

6.4.4.2 Description

From an algorithmic view, the Einstein notation is the most complicated module in this thesis. To simplify my explanation I choose EN as an abbreviation of the Einstein notation.

Whereas the single EN is easy to handle, the main problem in this part is the variability and therefore the amount of potential and legal notations. Generally, the EN is based on the classical binary arithmetic operators on complex access in the IR layer e.g. *IR_Multiplication(IR_ComplexAccess(...), IR_ComplexAccess(...))*. This example is pictured at code snipped [25] at line 15-17.

6.4.4.3 Chainability problem

One of the first problems at implementation is how a EN can be concatenated. E.g.:

```
1 Var: res : Real = t1[i,0] * t2[i,2] + t3[i,1]
```

Listing 26: ExaSlang4: Chained Einstein notation

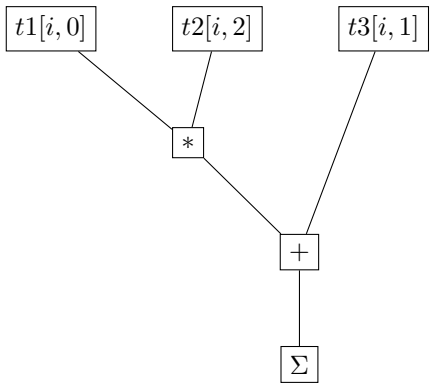

Figure 7: AST: example chained Einstein notation

The language structure of the Einstein notation [5.7] says that for an Einstein notation a binary (arithmetic) operator and two complex access with minimum an Einstein iterator [18] is necessary. Consequently, complex accesses are located on each leave [7].

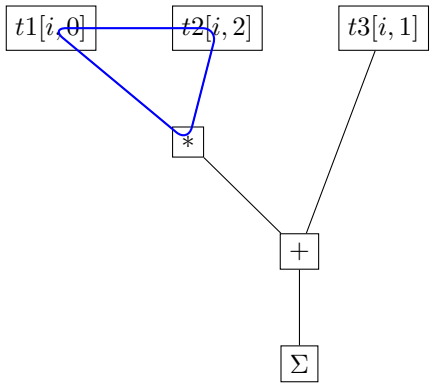

Figure 8: AST: example chained Einstein notation step 1

To determine the return type of the first EN-transformation, we need a deeper insight into the corresponding source data types. The left complex access $t1[i,0]$ is an access on a second-order tensor, where the Einstein iterator is placed on the first axis. The second access has the same properties, accordingly, an EN-transformation is possible.

To offer a further EN-transformation, which is displayed in step 2 [9], results of ENs also have to be a complex access type. Furthermore, they have to also have an Einstein iterator on the axis as the source accesses had. The problem is to discriminate between the source axis if the resulting complex access is on a first-order tensor because the inner data type does not imply any direction

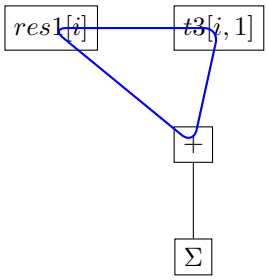

Figure 9: AST: example chained Einstein notation step 2

and axis. To treat this issue, I choose a transformed vector as the inherent return type of the concluded complex access, which holds the former Einstein iterator.

Here an illustration for the problem:

$$t1[i,2] * t2[i,0] * t3[i,2] * t4[i,0] \rightarrow res1[i] * res2[i] \tag{42}$$
$$\neq \tag{43}$$
$$t1[2,i] * t2[0,i] * t3[2,i] * t4[0,i] \rightarrow res1[i] * res2[i] \tag{44}$$

This example shows the reason why the return value has to be able to determine the former axis of the iterator.

The last step [10] of an EN is the summation over the transformation result.

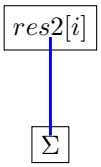

Figure 10: AST: example chained Einstein notation step 3

The result is a complex access data type and does not match to an EN, so it has to be transformed independently. The transformation [6.4.3.2] will be invoked if no not-transformed ENs were found.

6.4.4.4 The transformation in detail

Here the example $Var : res : Real = t1[i,0] * t2[i,2] + t3[i,1]$ from the last topic is used to describe the transformation. May $t1, t2, t3$ second order tensors, $a_{ij} \in t1, b_{ij} \in t2, c_{ij} \in t3$ elements of the tensors [11].

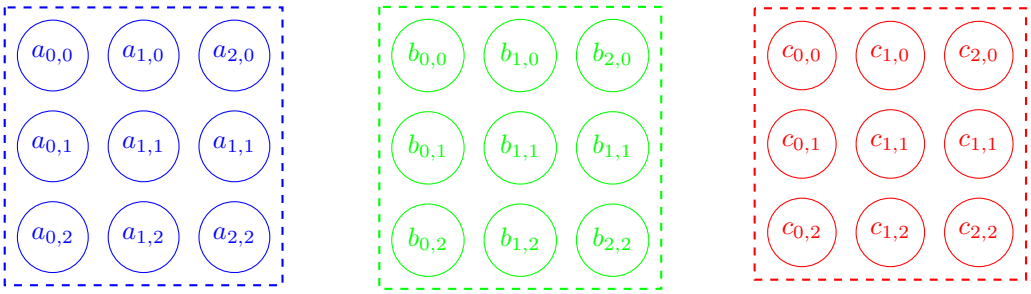

Figure 11: Detailed transformation of the Einstein notation

Now the first step [12] will be applied on the first two tensors: $res1 := t1[i,0] * t2[i,2]$, where $res1$ is a complex access on the resulting tensor. $res1$ will not be saved and transformed into target code, it is just an alias for the example.

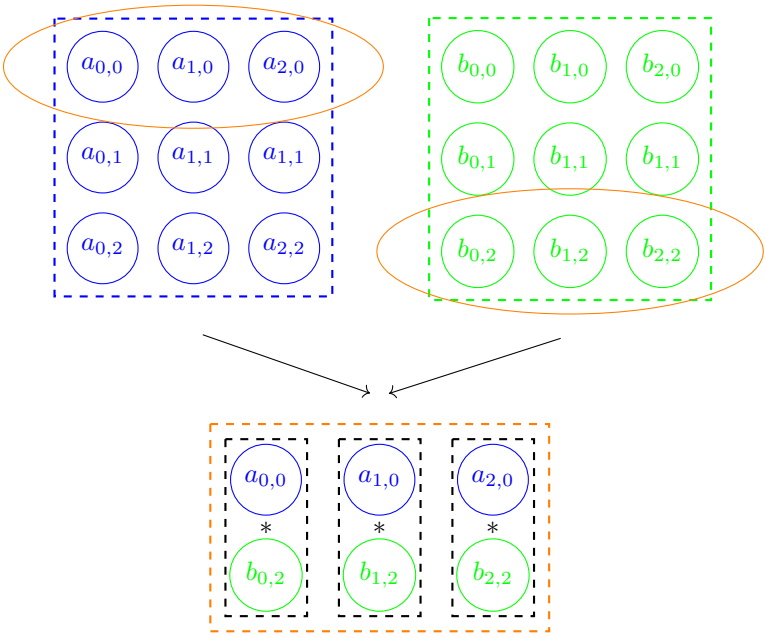

Figure 12: Detailed transformation of the Einstein notation step 1

Firstly, the indices of the accesses will be checked, the result tensor (or matrix) will be determined and created. Then, the transformation iterates over both source tensors. For each iteration, the left and right index will be calculated and implemented in the needed binary operator (e.g. $*$ in *IR_Multiplication*). Those binary operations will be added to the result tensor. After the result tensor will be filled, the complex access will be built, which has to be returned. At last, the result tensor will be added to the return value and the leftover indices including the used Einstein iterator will be copied to the index of the concluding complex access.

The second step [13] is almost similar to the first unless the transformation has to handle different inner data types. The left access is on the return value of a former EN, whereas the right complex access got variable access on the $t3$ tensor as in the first step. The transformation has to handle them both different because the right contains objects which can be copied, on the other side new high dimensional accesses to coordinates must be generated. For an easier transformation, I passed an additional parameter to the complex access constructor, which is called *deep*. The parameter is 0 if the access does hold a natural data type and not an artificial expression to a return value.

The last step [14] is the summation of concluding access. As indicated before, this is a part of the transformation *IR_ResolveComplexAccess*, which contains a branch that consumes the return of an Einstein notation. Therefore, the last transformation takes the array-like result complex access and orders into an *IR_Addition*.

6.4.4.5 Dissection of the resolve method

The resolve method in Einstein notations' transformation is the most significant single functionality implemented during the thesis.

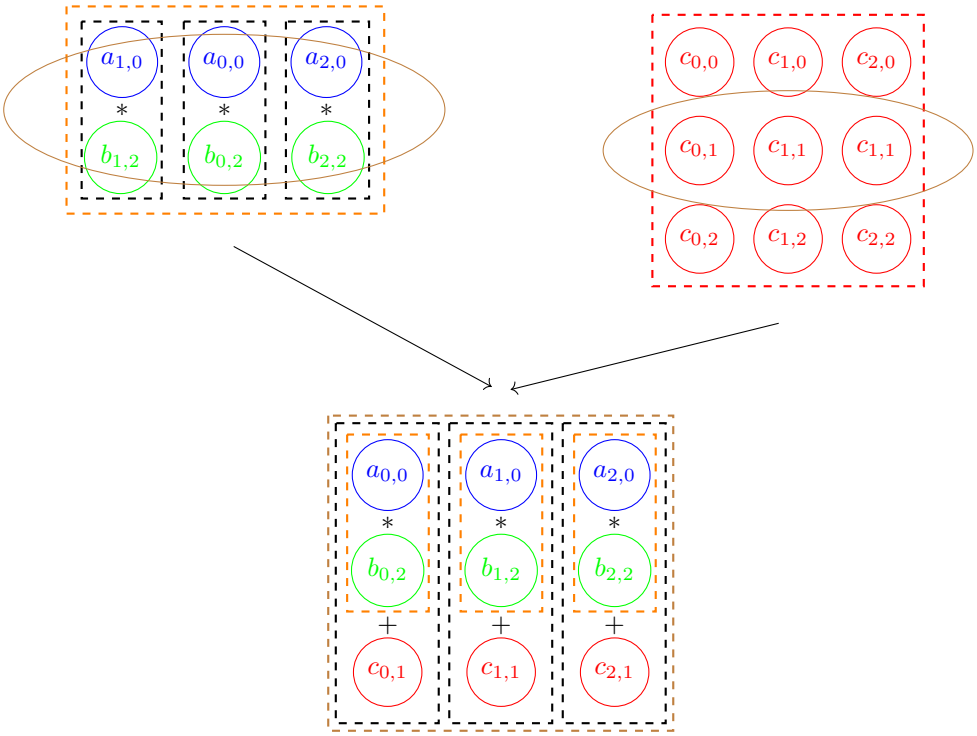

Figure 13: Detailed transformation of the Einstein notation step 2

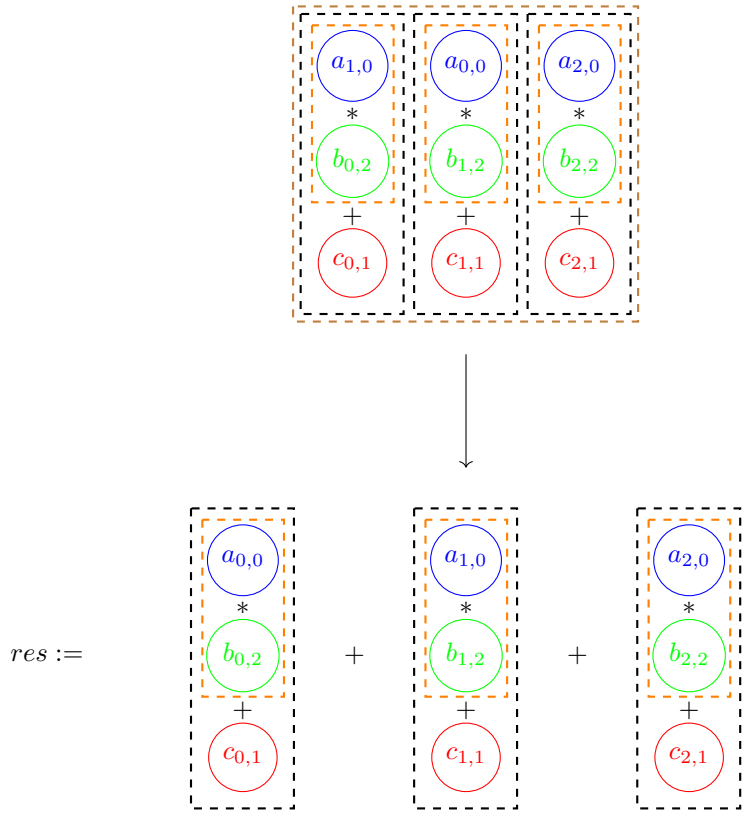

Figure 14: Detailed transformation of the Einstein notation step 3

The algorithm [2] shows the general steps of the method. Whereat, the primary transformation will also shown at the listings [27, 28]. Coarsely, the method can be separated into eight distinct

Algorithm 2: Eistein notation, resolve method

Data: input data: $access1 : IR\_ComplexAccess$, $access2 : IR\_ComplexAccess$, $operator :$ Char

Result: output data res

1. Case: Classical access
2. Construct necessary values
3. Check for edgecases, failurecases
4. Evaluate dimensionality
5. Map indeces in first and second access
6. Find indeces of the first in the second access
7. Test if the resulting dimensions of both tensors are correct
8. Main transformation [27, 28]

sections.

The first section returns a correct solution but not of an Einstein notation. The transformation matches a binary operator with two complex-access, which allows variable-access and concrete indices, not even einstein iterators. The second, third and fourth sections are constructive. Here where values for the calculation built and checked on their correctness. So this block determines if the given input accesses are correct and have proper form and datatype. The fifth, sixth and seventh sections map and control the iteration indices and contraction dimension. This part check where doubled einstein iterators are located at an index and look if they on the correct position. In this block, the resulting dimensionality and contraction will be determined and checked.

```
1  // the main transformation
2  contraction match {
3    // Contracts 1 order
4    case con if (con == 1) =>
5      val res_order : Int = List(order1 - contraction, order2 - contraction).max
6      res_order match {
7        case x if (x == 1) => // result first order
8        // ...
9        val ind : Int = ind_both.head
10       var res_tensor : IR_Expression = null
11       ind match {
12         // first dimension of the array a[ind,1]
13         case ind if (ind == 0) =>
14           res_tensor = IR_TensorExpression1(
15             IR_ResultingDatatype(tens_datatype1, tens_datatype2),
16             dims
17           )
18           for (i <- 0 until dims) {
19             order1 match {
20               case order if (order == 1) => index_left += i
21               // ...
22             }
23         // second dimension of the array a[1,ind]
24         case ind if (ind == 1) =>
25         // ...
26       } // end match ind
```

Listing 27: ExaStencils: IR Layer, Transformations Einstein notation

The eighth section contracts the input. For evaluating the correct contraction, several variables are needed. *contraction* contains the value of how many dimensions have to be contracted. The resulting tensor's order is stored at res_order, and *ind* contains the position of the doubled Einstein iterator. The deep parameter of complex-access shows if the access is a natural access or a return value of another Einstein notation.

The recently used example [12] can be used to display the values in the method. There is one doubled Einstein iterator named i, so *contraction* is equal to 1. Both input tensors are second-order tensors. So res_order must be $2 - 1$. The found Einstein iterator location is 0, so the first dimensionality must be contracted, and $ind = 0$. When looking at the branches at listing [27], the actual composition matches lines 4, 7 and 13. Between lines 14 and 22, the new tensor will be

created, and the indices of the left and right access will be calculated.

```
27        for (i <- 0 until dims) {
28          // ...
29          access1.deep match {
30          case deep1 if (deep1 == 0) =>
31            value_left = IR_HighDimAccess(
32              tens_access1, IR_ExpressionIndex(IR_IntegerConstant(index_left(i)))
33              )
34            )
35          case deep1 if (deep1 == 1) =>
36            access1.decl.initialValue match {
37            case Some(init) if init.isInstanceOf[IR_TensorExpression1] =>
38              value_left =
39                init.asInstanceOf[IR_TensorExpression1].expressions(index_left(i))
40              // ...
41          }
42          access2.deep match {
43            // ...
44          }
45          res_tensor match {
46            case res_tensor : IR_TensorExpression1 =>
47            operator match {
48              case op if (op == '*') =>
49                res_tensor.set(
50                  i,
51                  IR_Multiplication(value_left, value_right)
52                )
53              // ...
54            }
55          case res_tensor : IR_MatrixExpression  =>
56            // ...
57          }
58        } // end for
59        val decl = IR_VariableDeclaration(
60          res_tensor.datatype, "ct_" + access1.name + "_" + access2.name,
61          res_tensor
62        )
63        val res = IR_ComplexAccess(
64          decl.name,
65          decl,
66          None,
67          List(access1.mulDimIndex(ind)),
68          1
69        )
70        res
71    } // end match resorder
72    // Classic double contraction
73  }
```

Listing 28: ExaStencils: IR Layer, Transformations Einstein notation, part 2

The second part of the listing [28] combines the arithmetic operator and the left, respectively right accessed input. The combination will be saved at a previously calculated coordinate at the resulting tensor/matrix. In our example, both input accesses are on parsed tensors and not due to a former EN transformation. That means the *deep* parameter of both accesses is equal to 0, and the instance matches line 30. From line 59 until 69, the resulting *IR_ComplexAccess* will be created, which will be returned at line 70.

6.4.4.6 Review

Now it is time for a brief review of the Einstein notation:

- One line of ExaSlang4 produces one line of target code.
- No additional variables will be declared.
- The tensors' structure and the notation are known at the generation time so that complex operation can be pushed away from runtime.

- The module operates with variable access instead of concrete values, so it is usable in various functions.

In summary, these are relevant attributes for fast code, which is needed for use in scientific computing.

6.4.5 Contraction

As metioned before, the contraction is part of the Einstein notation, technically. So it does not have a distinct transformation. The first part of the EN transformation is to identify Einstein iterators and match to the correct transformation branch or abort if the access is not in the legal form. The (double) contraction always has the same form:

$$t1[a,b,c,d] * t2[c,d] \tag{45}$$
$$t3[a,b] * t4[a,b] \tag{46}$$

After checking the Einstein iterators on the axis, the tensors will be contracted. I decided to dissect the summand by functional programming form to enable each result to be calculated in one target code line at generation-time and be arithmetically optimized by the code generator. The strategy is focused on the goal, that every complex operations can be done at generation time and without saving intermediate steps.

6.4.6 Arithmetic operators

6.4.6.1 Transformation

```
object IR_ResolveTensorMultiplicationDivision
  extends DefaultStrategy("Resolve multiplication/divition with tensors") {
  // ...
  private def elemwiseMulTwoTensors1(
    m: IR_VariableAccess,
    n : IR_VariableAccess
  ) : IR_TensorExpression1 = {
    val tens1 = m.datatype.asInstanceOf[IR_TensorDatatype1]
    val tens2 = n.datatype.asInstanceOf[IR_TensorDatatype1]
    if (tens1.dims != tens2.dims) {
        Logger.error(
          "Elementwise multiplication two Tensor1: has different dimensionality, "
          + tens1.dims.toString + " != " + tens2.dims.toString
        )
    }
    val tmp = IR_TensorExpression1(
      IR_ResultingDatatype(tens1.datatype, tens2.datatype), tens1.dims
    )
    for (x <- 0 until tens1.dims) {
      tmp.set(x, IR_Multiplication(getElem(m, x, 0, Nil), getElem(n, x, 0, Nil)))
    }
    tmp
  }
  // ...
  def mulScalar(m : IR_Expression, n : IR_Expression) : IR_TensorExpression = {
    (m, n) match {
   // variable access
      case (m : IR_VariableAccess, n : IR_VariableAccess) if
        m.datatype.isInstanceOf[IR_TensorDatatype1] &&
        n.datatype.isInstanceOf[IR_ScalarDatatype]     => scalarMulTensor1(m, n)
      case (m : IR_VariableAccess, n : IR_VariableAccess) if
         m.datatype.isInstanceOf[IR_ScalarDatatype] &&
         n.datatype.isInstanceOf[IR_TensorDatatype1]    => scalarMulTensor1(n, m)
  // ...
```

Listing 29: ExaStencils: IR Layer, Transformations arithmetic operator part 1

```
  def elemMul(m : IR_Expression, n : IR_Expression) : IR_TensorExpression = {
    (m, n) match {
      case (m : IR_VariableAccess, n : IR_VariableAccess) if
        m.datatype.isInstanceOf[IR_ScalarDatatype] ||
        n.datatype.isInstanceOf[IR_ScalarDatatype]    => mulScalar(m, n)
      case (m : IR_TensorExpression, n : IR_VariableAccess) if
  // ...
  this += new Transformation(
    "resolution of multiplication/divition with tensors 2/3", {
    case call : IR_Multiplication if (
      // variable type / tensor
      (isFactor(call.factors(0)) && isTensor(call.factors(1))) ||
      // tensor / variable type
      (isTensor(call.factors(0)) && isFactor(call.factors(1))) ||
      // tensor / tensor
      (isTensor(call.factors(0)) && isTensor(call.factors(1)))
    ) =>
    if (call.factors.length < 2) {
       Logger.error("* must have two arguments")
    }
    if (call.factors.length > 2) {
      val factors = ListBuffer[IR_Expression](
        elemMul(call.factors(0), call.factors(1))
      )
      factors.appendAll(call.factors.drop(2).toList)
      IR_Multiplication(factors)
    } else {
      elemMul(call.factors(0), call.factors(1))
    }

    case call : IR_Division if (
      (isFactor(call.left) && isTensor(call.right)) || // variable type / tensor
      (isTensor(call.left) && isFactor(call.right)) || // tensor / variable type
      (isTensor(call.left) && isTensor(call.right)) // tensor / tensor
    )   =>  div(call.left, call.right)
  })
}
```

Listing 30: ExaStencils: IR Layer, Transformations arithmetic operator part 2

6.4.6.2 Description

Arithmetic operators on tensors have a similar structure, a transformation on the operator in the IR layer (e.g. *IR_Multiplication*), a match-case operator/method (e.g. *elemMul*) which selects implementation for the data types and an implementation part (e.g. *elemwiseMulTwoTensors1*) for the single type.

For this design, there are positive and also negative arguments. On the negative side, is the amount of code. Dissecting the code very early results in much more effort of methods and lines of scala code. In the moment of changing a simple datatype, lots of sources have to be surveyed as well. On the other hand, these methods are relatively small and homogenous, also basic data types, e.g. IR_Number, were rarely modified. Therefore, I decided to accept the code-effort for improved clearness and simplified debugging.

To apply operator-binding alike multiplication/division before addition/subtraction, I separated the transformations for both and iterate over them in the *IR_LayerHandler*. This procedure is displayed in the code snippet below. The transformations are placed in two loops. The inner loop for $*, /$ and the outer loop for $+, -$. The count of $3x3$ iterations was chosen pragmatically based on a handwritten equation's maximum recursive deep.

```
for (i <- 0 until 3) {
  for (j <- 0 until 3) {
    IR_ResolveTensorMultiplicationDivision.apply()
    IR_ResolveTensorReturnValues.apply()
  }
  IR_ResolveTensorAdditionSubtraction.apply()
  IR_ResolveTensorReturnValues.apply()
}
```

Listing 31: ExaStencils: IR Layer Handler, apply Multiplication/Division first

6.4.7 Other compile time functions

6.4.7.1 Transformation

```
object IR_ResolveTensorCompiletimeFunctions
  extends DefaultStrategy("Resolve special tensor functions") {

  this += new Transformation("resolution of built-in functions 2/2", {

    case call : IR_FunctionCall if (call.name == "dyadic")    =>
      if (call.arguments.length != 2) {
        Logger.error("dyadic() must have two argument")
      }
      dyadic(call.arguments(0), call.arguments(1))

  })
}
```

Listing 32: ExaStencils: IR Layer, Transformations other compiletime function

6.4.7.2 Description

The transformation of the other compile-time functions is analogue to the arithmetic functions. Despite the arithmetics, the other methods have an infix notation and will be converted out of an *IR_FunctionCall* by separating the name attribute.

In the layer handler, they have to apply before the arithmetics because they include constructor-like functions to illustrate the dyadic product.

The determinant will be transformed at compilation time because only the determinant for a second-order tensor with the dimensionality 3 is implemented. Later, the Laplace's expansion for more complicated tensors could be implemented, then this module will be dissectable between compile- and runtime.

6.4.8 Eigenvalues

6.4.8.1 Transformation

```
object IR_ResolveTensorRuntimeFunctions
  extends DefaultStrategy("Resolve special tensor functions") {

  def householder_step(
    I : IR_VariableAccess,
    Q : IR_VariableAccess,
    R : IR_VariableAccess,
    sign : IR_VariableAccess,
    alpha : IR_VariableAccess,
    v : IR_VariableAccess,
    i : IR_VariableAccess,
    q : IR_TensorDatatype2,
    dims : Int
  ) : IR_Scope = {
      // ...
  }

  def qrDecompHouseholder(
    A : IR_VariableAccess,
    res : IR_VariableAccess,
    Q : IR_VariableAccess,
    R : IR_VariableAccess
  ) : IR_Scope = {
    // ...
  }

  def eigenvalue(A: IR_Expression, res : IR_Expression) : IR_Scope = {
    // ...
  }

  this += new Transformation("resolution of built-in functions 2/2", {

    case IR_ExpressionStatement(call) if
    (call.isInstanceOf[IR_FunctionCall]) &&
    (call.asInstanceOf[IR_FunctionCall].name == "eigen") =>
      val call_res = call.asInstanceOf[IR_FunctionCall]
      if (call_res.arguments.length != 2) {
        Logger.error("eigen() must have two arguments")
      }
    eigenvalue(call_res.arguments(0), call_res.arguments(1))
  })
}
```

Listing 33: ExaStencils: IR Layer, Transformations eigenvalues

6.4.8.2 Description

The solver of the eigenvalue problem is the only runtime functionality that I have implemented for my thesis. The transformation returns a scope, which contains the whole solver code. Usually, this tactic will be taken if the function is encapsulated in another method or not often called in the main source code. The reason for this is that the scope including the code will be copied completely.

The function has thee sections: the method *eigenvalue*, which directly implements the QR-algorithm [15], the method *qrDecompHousholder* which executes a QR-decomposition and the *householder_step* that performs a Householder transformation for the decomposition. All these submethods constructs an individual scope, too.

7 Tests

Testing is a usual process in software development, indeed. At which tests were developed separately.

A unique characteristic of code generation tools is that they more or less test itself. Tests were developed in the source language, then they will be generated with the tool, and after that, the

Module	Desription
Constructors	Tests classical constructors, conversion and dyadic product
Arithmetic	Tests tensor arithmetic with all not deprecated datatypes
Access	Tests access with the bracket operator
EinsteinNotation	Tests diverse Einstein notations, inclusive, chained notations
HigherFunctions	Tests higher functions with are computed at compilation time
Eigenvalues	Tests small eigenvalue problems

Table 4: Table of test modules.

product must have an explicit output to runtime.

The tensor test is a complete test module, which tests all functions [7] of tensors. Each kind of tensor function is based in a single folder and will be generated separately. Tensor-tests construct tensors and use them in the described functionality.

8 Results and outlook

When I started with this thesis, it was unclear where problems could appear and which module costs time. I used the first quarter of my task, for the theoretical background, about tensors and ExaStencils and respectively ExaSlang. The second quarter, where to build the technical outlines, e.g. implementation of the datatypes and expressions, modification of the parser. The third and last quarter was used to write the thesis and implement detail functions. I also refactored the code structure, added the testing and the support of other datatypes, e.g. matrix and numbers. The most complicated individual parts were the bracket operators, the Einstein notation and the eigenvalue problem.

Often, by working with complex data types, I had to ask myself wether the problem dealt with was solvable at generation time or it was necessary to do it at runtime. A personal result is that every direct solvable problem type will not have to be calculated at runtime. This awareness is not new and follows directly out of the computability. So, what is new in this context. New is the code generation technology. Usual (target) programming languages, e.g. c++ are limited to abilities of the software developer. Complex data types will be manipulated by iterating over their items. If the calculation of the new values is more complicated some temporary variables were needed. All these impediments are not relevant for code generation.

E.g.: May $t1, t2$ second-order tensors with the dimensionality 3 and real as inner data type.

```
1 Var t3 : Tensor2< Real, 3 > = t1 + t2
```

Listing 34: ExaSlang4: Example for results

```
1 std::array<double, 9> t3;
2
3 for (int i = 0; i < t3.size(); ++i){
4   t3[i] = t1[i] + t2[i];
5 }
```

Listing 35: Usual C++: Example for results

```
using __tensor2_3_double_t = double[9];

__tensor2_3_double_t t3 {
  ((t1[0])+(t2[0])),((t1[1])+(t2[1])),((t1[2])+(t2[2])),
  ((t1[3])+(t2[3])),((t1[4])+(t2[4])),((t1[5])+(t2[5])),
  ((t1[6])+(t2[6])),((t1[7])+(t2[7])),((t1[8])+(t2[8]))
};
```

Listing 36: Generated C++: Example for results

In this simple example, which a second-order tensor was declared and gets the result of the addition of two second-order tensors as its initial value. There are some differences in the outcoming c++. At the typical c++ declaration and initialization are separated, another iteration-variable and the size of the array are crucial for the program. The generated c++ code was optimized by the generator. The addition operator was disintegrated and spread over the tensor items, so that each element can be computed in the initial expression and not in a loop. Comparably, the generated code is implicitly loop unrolled or quasi vectorised.

Potentially, further work may be settled at the complex access module. I see chances by implementing advanced indexing related to NumPy [Noab]

```
t3[0:2, 1] = t1[0:2, 0] * t2[0:2, 0]
```

Listing 37: Python Access: Example for results

At the current state, open indexing will be interpreted as an Einstein notation and summarized automatically. Actually, slicing and computing are separated, however later, they can be associated to simplify calculations and code.

```
    Var t3 : Tensor1< Real, 3 > = t1[0..2, 0] * t2[0..2, 0]
```

Listing 38: ExaSlang4: Outlook python-like slicing

A next possible topic for the future is the extension of the tensor syntax.

```
t1 = tensN{ 3; 4;
    [0,0,0,0] := 1.0,
    [1,0,0,0] := 1.0,
    [2,0,0,0] := 1.0,
    [0,1,0,0] := 2.0
}

t1_new = tensN{ 3, 4,
    [0,0,0,0] => 1.0,
    [1,0,0,0] => 1.0,
    [2,0,0,0] => 1.0,
    [0,1,0,0] => 2.0
}
```

Listing 39: ExaSlang4: Outlook tensor expression

Another syntax was primarily planned for the tensor expression, which is more near to this of the stencil definition. For fastening the implementation of the parser definition, the separators were changed to ; instead of ,, and => were altered to :=. In later versions of the tensor data type, it is meaningful to choose the original notation.

Concludingly, the tensor data type and its functionality can be used now in recently instantiated use cases, the linear elasticity, Maxwells' equation, or in other models of the rigid body physics. The models should be tested with an implementation containing tensors.

References

[Arb] Prof. Dr. Peter Arbenz. *Lecture-Notes to "Numerical Methods for Solving Large Scale Eigenvalue Problems", Chapter 4: The QR-Algorithm*. URL: http://people.inf.ethz.ch/arbenz/ewp/Lnotes/chapter4.pdf.

[Boe20] Fabian Boehm. "Refactoring and extensions of ExaStencils capabilities for matrix datatypes and operations". en. Bachelor. FAU-Erlangen-Nuremberg, chair of system simulation, Nov. 2, 2020.

[Fig] *Figure: second order tensor*. URL: https://upload.wikimedia.org/wikipedia/commons/4/45/Components_stress_tensor.svg (visited on 03/03/2020).

[For] Bryan Ford. "Packrat Parsing: Simple, Powerful, Lazy, Linear Time". en. In: (), p. 12.

[GP02] Murdoch J. Gabbay and Andrew M. Pitts. "A New Approach to Abstract Syntax with Variable Binding". en. In: *Formal Aspects of Computing* 13.3-5 (July 2002), pp. 341–363. ISSN: 0934-5043, 1433-299X. DOI: 10.1007/s001650200016. URL: http://link.springer.com/10.1007/s001650200016 (visited on 09/15/2020).

[Has06] Mehdi Hage Hassan. "Inertia tensor and cross product In n-dimensions space". In: *arXiv:math-ph/0604051* (Apr. 2006). arXiv: math-ph/0604051. URL: http://arxiv.org/abs/math-ph/0604051 (visited on 10/23/2020).

[Hol00] Gerhard A. Holzapfel. *Nonlinear Solid Mechanics: A Continuum Approach for Enineering: A Continuum Approach for Engineering*. Englisch. 1. Chichester ; New York: Wiley, Mar. 2000. ISBN: 978-0-471-82319-3.

[Its19] Mikhail Itskov. *Tensor Algebra and Tensor Analysis for Engineers: With Applications to Continuum Mechanics*. en. Mathematical Engineering. Cham: Springer International Publishing, 2019. ISBN: 978-3-319-98805-4 978-3-319-98806-1. DOI: 10.1007/978-3-319-98806-1. URL: http://link.springer.com/10.1007/978-3-319-98806-1 (visited on 09/13/2020).

[Klo00] jan willem Klop. "Term Rewriting Systems". In: (Aug. 2000).

[Kuc19] Sebastian Kuckuk. "Automatic Code Generation for Massively Parallel Applications in Computational Fluid Dynamics". Englisch. In: *Dr-Arbeit* (May 2019), p. 211.

[Noaa] *Expression (computer science)*. en. Page Version ID: 966695923. July 2020. URL: https://en.wikipedia.org/w/index.php?title=Expression_(computer_science)&oldid=966695923 (visited on 09/17/2020).

[Noab] *NumPy - Advanced Indexing - Tutorialspoint*. URL: https://www.tutorialspoint.com/numpy/numpy_advanced_indexing.htm (visited on 10/13/2020).

[Noac] *Partial Differential Equations in Physics*. en. Google-Books-ID: PFsDVARE4C0C. Academic Press, Jan. 1949. ISBN: 978-0-08-087309-1.

[Noad] *Statement (computer science)*. en. Page Version ID: 966755166. July 2020. URL: https://en.wikipedia.org/w/index.php?title=Statement_(computer_science)&oldid=966755166 (visited on 09/17/2020).

[Noae] *The Julia Programming Language*. URL: https://julialang.org/ (visited on 10/21/2020).

[Noo85] Robert E. Noonan. "An algorithm for generating abstract syntax trees". en. In: *Computer Languages* 10.3 (Jan. 1985), pp. 225–236. ISSN: 0096-0551. DOI: 10.1016/0096-0551(85)90018-9. URL: http://www.sciencedirect.com/science/article/pii/0096055185900189 (visited on 10/23/2020).

[San16] Wytler Cordeiro dos Santos. "Introduction to Einstein-Maxwell equations and the Rainich conditions". en. In: *arXiv:1606.08527 [gr-qc]* (June 2016). arXiv: 1606.08527. URL: http://arxiv.org/abs/1606.08527 (visited on 09/13/2020).

[Sca] *The Scala Programming Language*. URL: https://www.scala-lang.org/ (visited on 10/23/2020).

[Sch+14] Christian Schmitt et al. “ExaSlang: A Domain-Specific Language for Highly Scalable Multigrid Solvers”. en. In: *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. New Orleans, LA, USA: IEEE, Nov. 2014, pp. 42–51. ISBN: 978-1-4673-6757-8. DOI: 10.1109/WOLFHPC.2014.11. URL: http://ieeexplore.ieee.org/document/7101662/ (visited on 01/12/2020).

[Sha04] Ruslan Sharipov. “Quick introduction to tensor analysis”. In: *arXiv:math/0403252* (Mar. 2004). arXiv: math/0403252. URL: http://arxiv.org/abs/math/0403252 (visited on 10/23/2020).

[Tem01] Roger Temam. *Navier-Stokes equations: theory and numerical analysis*. en. Providence, R.I: AMS Chelsea Pub, 2001. ISBN: 978-0-8218-2737-6.

[VQEW18] https://commons.wikimedia.org/w/index.php?curid=27645908 Von Quartl Eigenes Werk CC BY-SA 3.0. *Figure Dyadisches Produkt*. de. Page Version ID: 176713475. Apr. 2018. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/8/8d/Matrix_multiplication_qtl3.svg/1920px-Matrix_multiplication_qtl3.svg.png (visited on 03/03/2020).

[Åh02] K. Åhlander. “Einstein summation for multidimensional arrays”. en. In: *Computers & Mathematics with Applications* 44.8 (Oct. 2002), pp. 1007–1017. ISSN: 0898-1221. DOI: 10.1016/S0898-1221(02)00210-9. URL: http://www.sciencedirect.com/science/article/pii/S0898122102002109 (visited on 10/23/2020).