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

**Lehrstuhl für Informatik 10 (Systemsimulation)**

# Automated Dependency Analysis and Scheduling of Transformation Strategies in ExaStencils

Maik Haase

Masterthesis

# Automated Dependency Analysis and Scheduling of Transformation Strategies in ExaStencils

Maik Haase

Masterthesis

| | |
|---|---|
| Aufgabensteller: | Prof. Dr. H. Köstler |
| Betreuer: | Dr. Ing. Sebastian Kuckuk |
| Bearbeitungszeitraum: | 01.10.2019 – 05.08.2020 |

**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 Masterthesis einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 4. August 2020 .....

# Abstract

With the rising need for efficient high-performance software in scientific computing, a variety of different approaches are created to reduce the development cost of such software, as well as increase its portability between different architectures. The ExaStencils project provides a code generation framework and introduces a set of four domain-specific languages aimed towards STEM professionals with varying background knowledge. The core of the code generation framework is given by the ExaStencils Compiler which translates source from one of the input DSLs to a C++ implementation by a series of transformation strategies. In the scope of this thesis, an approach is developed to automate the strategy handling. Therefore, a unifying interface for strategy execution is proposed in form of a scala trait, and existing strategies are tailored to the introduced execution pattern. Besides, a strategy scheduling implementation is proposed based upon topological sorting, according to the strategies inherent dependency hierarchy. Furthermore, resolving the dependency structure is attempted. In order to narrow the search space of all possible permutations, a heuristic is proposed that introduces neighbourhoods to create permutations locally in a reduced interval. Underlying dependencies are finally resolved by performing compilations with example problems using the created strategy permutations. Therefore, a stepwise analysis approach is implemented to compare incrementing neighbourhoods with a reference run. The shown approaches provide a suitable method to resolve dependencies of strategies within proximity.

# Kurzfassung

Aufgrund des steigenden Bedarfs an effizienter Höchstleistungssoftware und der damit verbundenen hohen Entwicklungskosten gibt es verschiedene Ansätze die Erstellung entsprechender Programme zu vereinfachen. Einen dieser Ansätze bildet das ExaStencils Projekt, das mithilfe eines Codegenerators und eigens entwickelter, domänenspezifischer Programmiersprachen die Entwicklung neuer Programme erleichtern, sowie deren Optimierung automatisieren möchte. Die Kernaufgabe kommt dabei dem ExaStencils Compiler zu der in einer Reihe von Transformationsstrategien den Quellcode in eine C++-Umsetzung übersetzt.

Im Rahmen dieser Arbeit wird ein Ansatz entwickelt um die unterschiedlichen Transformationsausführungen zu vereinheitlichen und die Strategiehandhabung zu automatisieren. Zu diesem Zweck wird ein Scala Trait eingeführt und der Strategieaufruf auf ein strategy.apply Muster festgesetzt. Folgend wird ein topologisches Sortierverfahren implementiert, dass die verschiedenen Strategien mit Bezug auf bestehende Abhängigkeiten ordnet.

Im zweiten Teil dieser Arbeit wird ein Lösungsansatz vorgestellt um die Abhängigkeiten zwischen den Strategien aufzulösen. Dafür wird eine Methodik entwickelt, die die Abhängigkeiten aus unterschiedlichen Strategieanordnungen und dem Ergebnis des Übersetzungsvorgangs ableitet. Zwingend notwendig muss vorher ein Ansatz entwickelt werden, um den Suchraum aller möglich Strategieanordnungen zu reduzieren. Im Rahmen dieser Arbeit geschiet dies über die Einführung von Permutationsintervalen, die die gesamte Strategieanordnungen in kleinere Sequenzen unterteilt und lokal modifiziert. Die gezeigten Verfahren erweisen sich als sinnvolle Ansätze um Teile der Abhängigkeitshierarchie zu ermitteln.

# Contents

# List of Figures

# Listings

# List of Tables

# 1 Introduction

Many of today's scientific problems take the form of partial differential equations (PDEs) and, while some PDEs can be solved analytically, the majority relies on computational methods. In order to solve such a problem, a variety of numerical methods have been established. The general concept behind these methods is to project the continuous problem onto a discretized grid and iteratively refine the solution until certain criteria are met. In reality, developing software that efficiently implements a numerical solver for a given problem is a demanding task and faces various challenges. The developer of such a program needs to have profound knowledge about the applications problem domain, whether it being computational fluid dynamics (CFD), computational optics, molecular dynamics, or else. For every problem in this variety of topics exist a different optimal implementation and a different optimal combination of numerical solvers that the developer needs to oversee. Unfortunately, due to the nature of this approach and the need of solving a problem on every grid point of a discretization, an optimized and efficient implementation is necessary. More so, since the computational processing power being needed to solve intriguing problems with reasonable resolution and accuracy surpasses that of modern computers as they are found in most households today. Therefore, programs that aim to solve or simulate scientific problems and systems are created to run on modern-day supercomputers. Such a supercomputer typically consists of a multitude of processing nodes, each of which a combination of different processors such as central processing units (CPUs) and graphics processing units (GPUs). In order to achieve supercomputing performance, an application needs to be fine-tuned to all advantages and disadvantages of the processing units and their specific architecture within the supercomputer. Since the number of processes can exceed 10.000.000 as in the Sunway Taihu Light[1] and the power usage might exceed 28 MW (Fugato, RIKEN[2]) it is crucial to create optimized code and reduce the program's runtime to a minimum. Consequently, the engineered program must not only be optimized concerning the problem domain but also with the target supercomputer's architecture in mind. Hence, the developer additionally needs profound knowledge of computer science to make use of all parallel computing capabilities that the target system offers.

One approach to mitigate these challenges and to enable interested science, technology, engineering and mathematics (STEM) professionals to use these computational methods is the introduction of a code generation framework.

Such a code generation framework allows to internalize otherwise required expertise and alleviates entry barriers for professionals with different background knowledge. Also, by defining individual syntax elements, the problem specification can be further abstracted from their computational representation and thus tailored to the problem domain.

[1] https://www.top500.org/system/178764/
[2] https://www.top500.org/system/179807/

The ExaStencils[1] project is precisely taking this approach; a code generation tool built in Scala[3] to create multigrid solvers that incorporate machine and domian knowledge into C++ code supporting parallelism through openMP[4], MPI[5], and CUDA[6].

[3] https://www.scala-lang.org/
[4] https://www.openmp.org/
[5] https://www.open-mpi.org/
[6] https://docs.nvidia.com/cuda/index.html

# 2 Background

## 2.1 Introduction to ExaStencils

ExaStencils is a code generation framework aimed towards STEM professionals providing features to generate optimized high-performance code. Instead of requiring specialized knowledge of supercomputers and parallelization techniques, the framework provides an easy-to-use interface to specify the problem. The problem domain in ExaStencils' context is limited to block-structured grids with geometrical multigrid solvers whose kernels can be expressed as stencil codes [2]. As an interface to the user, ExaStencils introduces a set of four domain-specific languages (DLS), ExaSlang. Given the variety of background knowledge in the target user space, those four different languages provide different levels of abstraction and complexity in the specification possibilities of the problem [3]. Finally, to conclude the ExaStencils project, the framework comes with a source-to-source compiler or code generator, written in Scala and built atop the term rewriting code transformation framework Athariac [4].

## 2.2 ExaSlang

The ExaStencils language, ExaSlang, is an external domain-specific language (DSL) tailored to provide an easy-to-use interface for the specification of geometrical multigrid solvers. Since numerical solvers for PDEs are applicable and needed in a broad spectrum of scientific and engineering topics, the background knowledge and expectations of possible users vary. ExaSlang, therefore, identifies three categories of user groups with different expectations regarding such a language. An abstract language with simple features to specify mathematical formulations of the problem as represented by natural scientists and engineers. Mathematicians, who are more intrigued by exploring the effects of different numerical solvers and changes to the underlying multigrid concepts. And finally, computer scientists, who are used to low-level languages and their features, with interests in the software implementation details such as parallelism and communication or memory access patterns. In order to cater to every user groups needs, ExaSlangs is designed as a set of four different languages with different levels of abstractions, in the following also referred to as Layer I - IV or ExaSlang I - IV. Additionally, ExaSlang provides an interface for target platform-specific descriptions, as well as auxiliary information for more control over the code generation process.

### 2.2.1 Language Elements

ExaSlang provides most language features that users are used to in other programing languages. Regarding data types, ExaSlang differentiates three major groups: Simple data

types such as *Integer* for natural numbers, *Real* for floating-point numbers, *String*, *Boolean*, and *Unit*; aggregate data types like *Vector* or *Complex* for complex numbers. Additionally, ExaSlang contains algorithmic data types, *Stencil* and *Field*, that serve a special purpose in the definition of the multigrid solver. For variable declaration, ExaSlang implements a syntax similar to Scala. Variables are declared by either the keyword *Variable* or *Var* followed by the type definition. As in Scala, the type is defined by a colon followed by the corresponding keyword. Note that in ExaSlang only simple and aggregate data types are applicable for variable declaration. ExaSlang also provides constant variables. These are declared similarly; however, the keyword *Value* or *Val* finds use instead. ExaSlang *Values* must be initialized after declaration.

*Fields* are conceptualized differently in each layer but can be viewed as a representation for a mathematical function in Layer I and as its discretization for lower layers. Essentially, *Fields* act as data containers that can be used, for example, as the right-hand side (RHS) of the PDE, as the vector of unknowns to be solved for, as a (temporary) result of a computation, among others.

In order to specify the whole PDE, *Operators* are needed in combination with *Fields*. For Layer I, ExaSlang *Operators* resemble mathematical operators and, similar to the concept of *Fields*, *Operators* are discretized as a matrix for lower layers. Regarding ExaStencils' restrictions, these matrices must be representable as *Stencils*. The *Stencils* values can either be specified using an offset regarding each grid point or by using the keywords *NORTH*, *WEST*, *SOUTH*, *EAST*, and *CENTRAL*. *Operators* and *Stencils* are used to implement, for example, the smoother, the prolongation, or the restrictions of the multigrid algorithm.

In consideration of developing whole multigrid solver applications, ExaSlang provides the possibility to define user-specified *Functions* and allows for branching with conditions as well as for loops. *Functions* are defined similarly to common programming languages. The definition is started using the *Function* keyword followed by an identifier, an optional list of parameters, and the return type. If a *Function* does not return any value, the data type *Unit* must be specified.

Conditions follow the known *IF-ELSE* syntax, however, loops are implemented differently compared to other programming languages. Syntactically, loops start with the keyword *repeat _ times* in case of a traditional for-loop or *repeat until _* in case of a while-loop implementation. In addition, ExaSlang offers the *loop over* keyword in order to loop over a computational domain, e.g. *Field*.

To conclude the list of language features that are needed to build a multigrid application, most objects in ExaSlang can be *leveled*. Leveling maps the *Function*, *Field*, *Operator*, *Stencil*, or else to the hierarchical structure of the multigrid algorithm. Leveled objects use the keywords *coarser*, *coarsest*, *finer*, *finest*, and *current*. Additionally, levels can be specified by enumerating and using ranges. Combinations of keywords and numerals, as well as excluding certain levels from ranges, are also supported.

More information on the topic of ExaSlangs language features can be obtained from Schmitt et al. [3] or Kuckuk [5].

### 2.2.2 ExaSlang's Layer approach

As stated above, ExaSlang is implemented as a set of four languages with varying levels of abstractions corresponding to the target user group's expectations and knowledge. ExaSlang I and IV are designed to work solely and allow for the generation of an entire multigrid application. In contrast, Layer II and Layer III codes complement each other in such a way that ExaSlang II contains the problem specification and discretization details and ExaSlang III allows the definition of the solver's implementation. How the layers relate to each other is depicted in Figure 2.1. The following sections briefly describe Layers I-IV using the two-dimensional Poisson equation as an example. For more information about ExaSlangs layers and their implementation specifics please refer to the dissertation of S.Kuckuk [5].

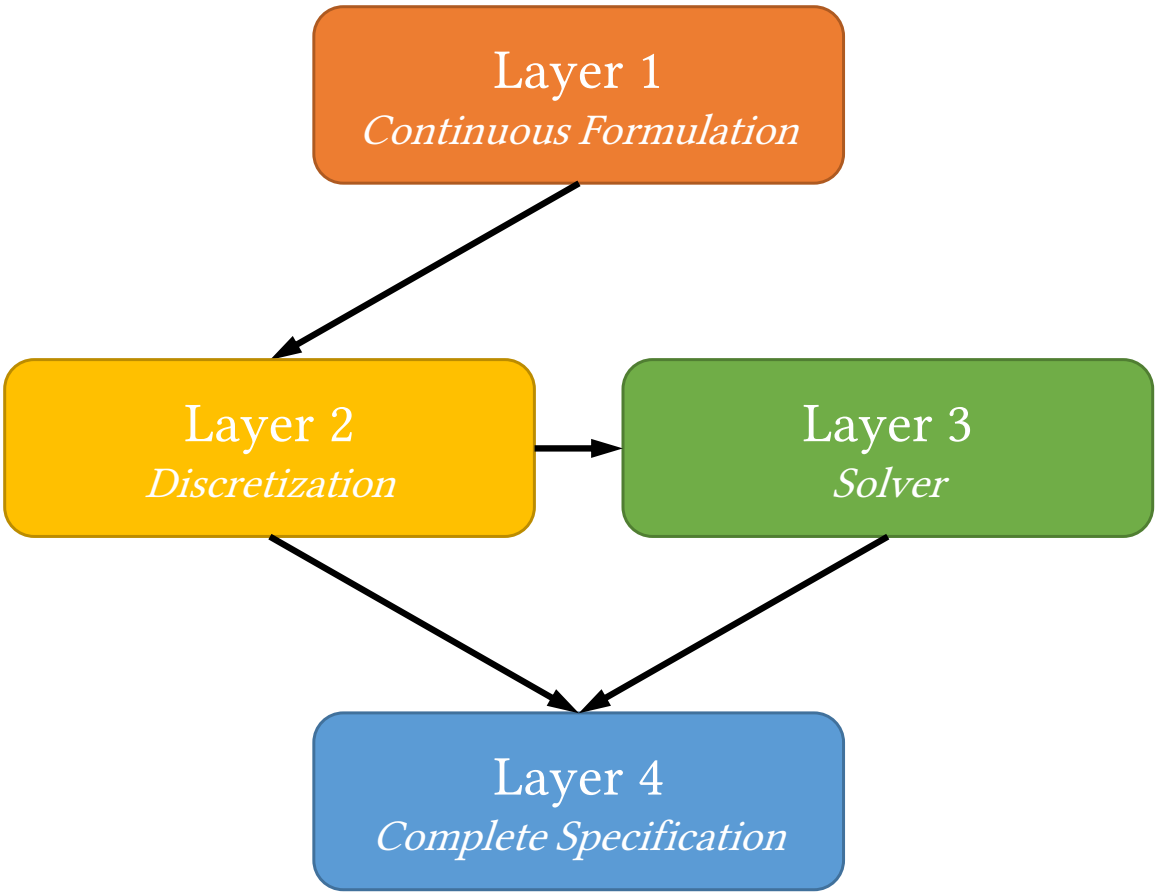



**Figure 2.1:** Relation between ExaSlang layers. Kuckuk [5]

#### ExaSlang I

Layer I strives to provide an easy-to-use way for natural scientists and engineers to create simulation software. Its syntax is based on LaTeX to ensure familiarity with the targeted user group. Therefore, Unicode symbols are supported. The goal of ExaSlang I is to easily copy and paste a problem setup from a paper and generate a working output program.

A correct problem specification begins with the specification of the domain as displayed in 2.1. ExaSlang I restricts the domain specification to those, which can be expressed as a cartesian product using intervals. Since Layer I's syntax is designed to be close to LaTeX's, Unicode characters such as Ω and × are allowed as well as their LaTeX representation *\Omega* and *\times*. This continues for Unicode characters that are listed throughout code examples of ExaSlang I.

```
1    Domain $Ω$ = (0,1) x (0,1)
```

**Listing 2.1:** Domain definition in ExaSlang I

Next, the underlying PDE and boundary conditions must be stated. In ExaSlang I this is done using the continuous formulation of the PDE. The discretization occurs during the code generation step to ExaSlang II. As containers for the starting and boundary conditions, ExaSlang I offers *Fields* (2.2).

```
1    Field $u ∈ Ω = 0.0 $
2    Field $u ∈ ∂ Ω = cos(π x) - sin(2π y) $
3    Field $f ∈ Ω = π^2 cos(π x) - 4π^2 sin(2π y) $
```

**Listing 2.2:** Definition of starting and boundary condition in ExaSlang I

*Operators* in ExaSlang I resemble mathematical operators and act similarly. A sample definition can be seen in Listing 2.3

```
1    Operator  $op = - \nabla $
2    Operator  $op = -(\delta\_\{xx\} + \delta\_\{yy\}) $
```

**Listing 2.3:** Operator declaration supports Unicode and LaTeX notation

Finally, the PDE can be specified as a combination of the given *Fields* and *Operators*. ExaSlang I reserves the Equation keyword for this purpose (2.4).

```
1    Equation $uEq: op  \cdot  u  = f $
```

**Listing 2.4:** Final definition of the PDE

All keywords in ExaSlang I can be omitted since their definition is unique.

### ExaSlang II

ExaSlang II is similar to ExaSlang I in the way simulation parameters are set up. However, Layer II shifts from the continuous representation of ExaSlang I to a discretized expression of *Operators* and *Fields*. The restriction on the selection of the computation domain is carried over, but the syntax is changed slightly. Instead of using a cartesian product of intervals, the lower left and upper right corner points are demanded. Listing 2.5 shows the ExaSlang II definition of the *Domain* and *Fields* equivalent to the example given for Layer I.

```
Domain global from [0,0] to [1,1]
Field Solution with Real on Node of global = 0.0
Field Solution on boundary = (cos (PI * vf_boundaryPos_x)
                           - sin (2.0 * PI * vf_boundaryPos_y))
```

**Listing 2.5:** ExaSlang II definition of a domain and fields.

In addition to these minor differences moving from Layer I to Layer II, ExaSlang redefines the implementation of *Operators*. The continuous formulation and resemblance to mathematical operators are ommited for a discretized notation. Listing 2.6 shows the discretized implementation of the Laplace operator and uses keywords as well as offsets for corresponding grid points.

```
Operator Laplace@finest from Stencil{
 center  =>   2.0/(vf_gridWidth_x**2)
            + 2.0/(vf_gridWidth_y**2)
 east    =>  -1.0/(vf_gridWidth_x**2)
 west    =>  -1.0/(vf_gridWidth_x**2)
 [ 0, 1] =>  -1.0/(vf_gridWidth_y**2)
 [ 0,-1] =>  -1.0/(vf_gridWidth_y**2)
}
```

**Listing 2.6:** Laplace operator in ExaSlang II

*Operators* that perform the mapping between different multigrid levels, namely *Restriction* and *Prolongation*, or any other interpolation operator, can be specified in a similar fashion. However, ExaSlang offers to construct default *Restriction* and *Prolongation* operators during the code generation. Additionally, ExaSlang II provides *Stencil Templates* that allow for the definition of *Stencils* with varying coefficients. More in-depth detail to Layer II's *Operators* can be found in the dissertation of S. Kuckuk [5].

### ExaSlang III

ExaSlang III's target user group is mainly interested in the solver's implementation and possible variations. As such, it is designed to offer freedom in the specification of the numerical solvers and details of the multigrid algorithm. *Fields* and *Operator* can be created similar to Layer II, however, it is possible in Layer III to copy previously generated *Fields* and adapt those. For the specification of the solvers, ExaSlang III allows for user-defined functions. Listing 2.7 shows an ExaSlang III definition of a Gauss-Seidel step.

```
Function GaussSeidel@all{
  Solution += (diag_inv(Laplace)
               * (RHS - Laplace * Solution))
}
```

**Listing 2.7:** Function definition in ExaSlang III

The syntax follows MATLAB to ensure familiarity with the target user group. Making use of user-defined functions allows an ExaSlang III developer to implement a multigrid solver. For users, who are satisfied with a default multigrid implementation with only slight adap-

tations, Layer III provides features to generate a multigrid solver according to particular user-specified parameters. The resulting implementation can afterwards be altered by ExaSlangs's *solver stages*. This enables the user to replace *Operators* and *Fields* with custom implementations or to add functionality to any given phase of the multigrid algorithm at the desired level.

### ExaSlang IV

ExaSlang IV presents the apex of ExaSlang's DSL set. As it targets at computer scientists, it builds the least abstract layer. ExaSlang IV syntax is influenced by C++ and in addition to all previous layer's functionality, parallelism features such as communication and memory access patterns are exposed to the user. Hereby comes an extension to *Field* declaration: *Layouts*. *Layouts* hold information about the *Field's* data type and structure. For example, information about the introduction of a ghost layer, and whether or not data will be communicated between threads can be defined, see Listing 2.8.

```
1  Layout NodeWithComm< Real, Node >@all {
2    duplicateLayers = [1, 1] with communication
3    ghostLayers     = [1, 1] with communication
4  }
5
6  Field Solution< global, NodeWithComm, 0.0 >@(all but finest)
```

**Listing 2.8:** Definition of a Field and a corresponding Layout

Conceptually, ExaSlang IV changes the focus to operations on the nodes of the underlying grid. This becomes apparent in the definition of *Operators* in the form of *Stencils* and *Functions*. Listing 2.9 displays the definition of a Gauss-Seidel step in ExaSlang IV. In contrast to Listing 2.7, the iteration space must be specified using the *loop over* keyword.

```
1  Function GaussSeidel@all {
2    loop over Solution{
3      Solution += ( diag_inv(Laplace)
4                  * (RHS - Laplace * Solution))
5    }
6  }
```

**Listing 2.9:** Function definition in ExaSlang IV

The entry point for every code generation in ExaStencils finally is the *Function Application* in ExaSlang IV(2.10).

```
Function Application {
  // init
  startTimer ( "setup" )
  initGlobals ( )
  initDomain ( )
  initFieldsWithZero ( )
  initGeometry ( )
  InitRHS@finest ( )
  apply bc to Solution@finest
  stopTimer ( "setup" )

  // solve
  startTimer ( "solve" )
  Solve@finest ( )
  stopTimer ( "solve" )

  // de-init
  if ( !getKnowledge ( 'testing_enabled' ) ) {
    printAllTimers ( )
  }
  destroyGlobals ( )
}
```

**Listing 2.10:** ExaSlang application as entry point for C++ code generation

A complete ExaStencils setup requires in addition to the ExaSlang source files further information. These are provided in the form of three separate config files for platform-specific information, compilation settings such as e.g. output path and application name, and additional parameters for the generated solver. Listings 2.11 - 2.13 shows an example for such *.platform*, *.settings*, and *.knowledge* files, however, their specifics will not be discussed further in the scope of this thesis.

Code transformation between *Layers*, and the data structure during code generation, called *Intermediate Representation* or *IR*, are discussed in section 2.4.

```
targetOS                = "Linux"
targetCompiler          = "GCC"
targetCompilerVersion   = 5
targetCompilerVersionMinor = 4
```

**Listing 2.11:** ExaStencils platform specification for a linux system.

```
1   user                = "Guest"
2   configName          = "2D_FD_Poisson_fromL4"
3   basePathPrefix      = "./Poisson/"
4   l4file              = "$configName$.exa4"
5
6   debugL1File         = "../Debug/$configName$_debug.exa1"
7   debugL2File         = "../Debug/$configName$_debug.exa2"
8   debugL3File         = "../Debug/$configName$_debug.exa3"
9   debugL4File         = "../Debug/$configName$_debug.exa4"
10
11  htmlLogFile         = "../Debug/$configName$_log.html"
12  outputPath          = "../generated/$configName$/"
13
14  produceHtmlLog      = true
15  timeStrategies      = true
16
17  buildfileGenerators = { "MakefileGenerator" }
```

**Listing 2.12:** ExaStencils settings specification.

```
1   dimensionality                  = 2
2
3   minLevel                        = 0
4   maxLevel                        = 8
5
6   discr_type                      = "FiniteDifferences"
7
8   // omp parallelization on exactly one fragment in one block
9   import '../lib/domain_onePatch.knowledge'
10  import '../lib/parallelization_pureOmp.knowledge'
```

**Listing 2.13:** Knowledge configuration file in ExaStencils

## 2.3 Compilers Fundamentals

Designing a programming language always demands an accompanying program to interpret or translate the newly introduced syntax. Such a program that creates machine instructions from a given language source is called a compiler.

In general, compilers transform from a source language to a target programming language. Frequent examples are the transformation from higher programming languages to less expressive ones, such as machine instructions or byte code. Another use case is the translation from one high-level programming language to another. The ExaStencils code generator represents the latter form, also known as source-to-source compiler.

Besides the translation, the tasks of the compiler also include optimization of the code to emit an efficient output program. In this context, the definition of efficient may vary according to the program's use case. Some possible optimization targets include the size of the output program, the run time of the final application, or lower power consumption.

In this chapter a brief introduction to the general compilation process is given, however, this description is not all-embracing and there exist compilers that endorse different approaches.

### 2.3.1 Compilation Workflow

The compilation process can roughly be separated into two main phases, code analysis and code generation. During the analysis phase, the structure of the input code as well as the logic behind it are examined, and a representation of the input code is created to work on during the compilation. Intertwined with the later steps of the process are machine-independent optimizations. Afterwards, machine-dependent optimizations are performed to tailor the program to the target system architecture before the final code is generated and emitted.

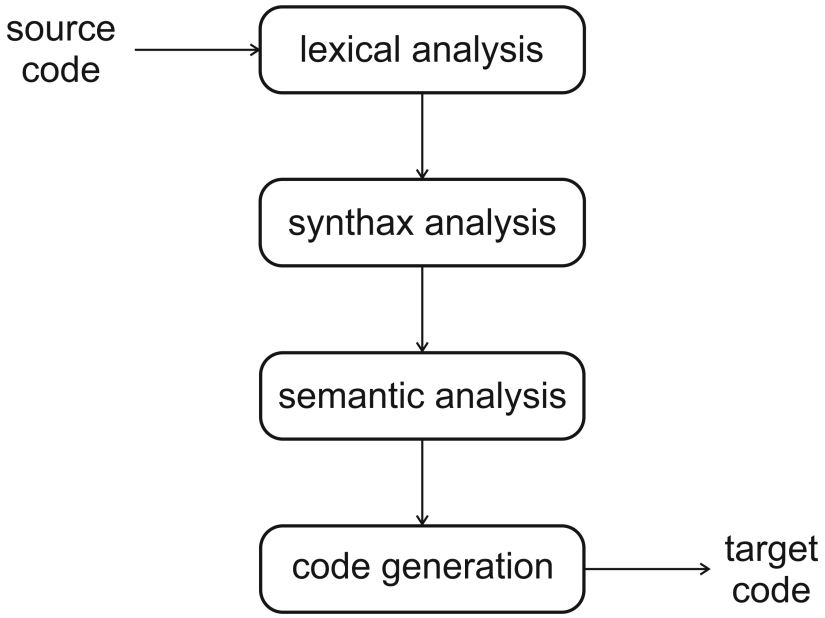



**Figure 2.2:** Phases of compilation. Modified according to Watson. [6]

#### Lexical Analysis

The compilation process begins with a text file in the desired programming language. The first task in the translation process is to extract and distinguish the different symbols of the source, some of which are important to reveal the program structure, others, e.g. user comments and whitespace, can be neglected during the compilation. This preparation of the input is called the lexical analysis and is performed by the *Lexer*.

The *Lexer* processes the input file and bundles it into the basic syntactic components of the source language. These components represent the language's elemental features, for example numerals, operators, and keywords bound to the language. The combination of a source code symbol and its corresponding syntactic components is called a *Token* and the entirety of *Tokens* provides the basis for the compilation process. Typical *Token*, as they

could be identified in a compiler, are exemplified in Table 2.1 according to the code snippet shown in Listing 2.14.

```
1           while (i < 100){
2           sum += vector[i];
3           i++;
4           }
```

**Listing 2.14:** Snippet of a while-loop in C

| | |
|---|---|
| identifiers | i, sum, vector |
| integer constants | 100 |
| keywords | while |
| operator | <, +=, ++ |
| punctuation | (), {}, [ ], ; |

**Table 2.1:** Matching of example lexer tokens of a while-loop implementation in C.

Once the essential components of the source code are extracted, the next step of the analysis focusses on giving structure to the *Token* stream.

### Syntax Analysis

The task of the syntactic analysis, also called *Parsing*, is to group the incoming *Token* conforming to the syntax rules of the source language, thus, extracting the logical structure of the given program. During this step, logical units are identified within the pattern of the incoming *Token* such as statements, expressions but also functions, loops, or variable declaration.

In order to demonstrate the parsing process, a set of syntax rules for the while-loop shown above, expressed in the Backus-Naur Form (BNF) is assumed. For simplicity, punctuation and loop body are neglected.

$$
\begin{array}{lll}
< loop > & ::= < whileKeyword > & + < condition > \\
< condition > & ::= < identifier > & + < comparison > \\
< comparison > & ::= < comparisonOperator > & + < integerConstant >
\end{array}
$$

According to these rules, a parser can match and replace patterns found in the token stream.

$$
\begin{aligned}
&while\ i\ <\ 100\\
&<whileKeyword>\ i\ <\ 100\\
&<whileKeyword><identifier><\ 100\\
&<whileKeyword><identifier><comparisonOperator>\ 100\\
&<whileKeyword><identifier><comparisonOperator><integerConstant>\\
&<whileKeyword><identifier><comparison>\\
&<whileKeyword><condition>\\
&<loop>
\end{aligned}
$$

Typically in modern compilers, for further processing of the program, the code is transformed into an internal representation in form of a concrete syntax tree (CST)(Figure 2.3).

As programming languages are more complex, the set of rules given here is utterly simplified. Therefore, the generated trees tend to hold more information, a lot of which is redundant. Therefore, the tree will be reduced before continuing with the next step and a more abstract version is obtained: the abstract syntax tree (AST).

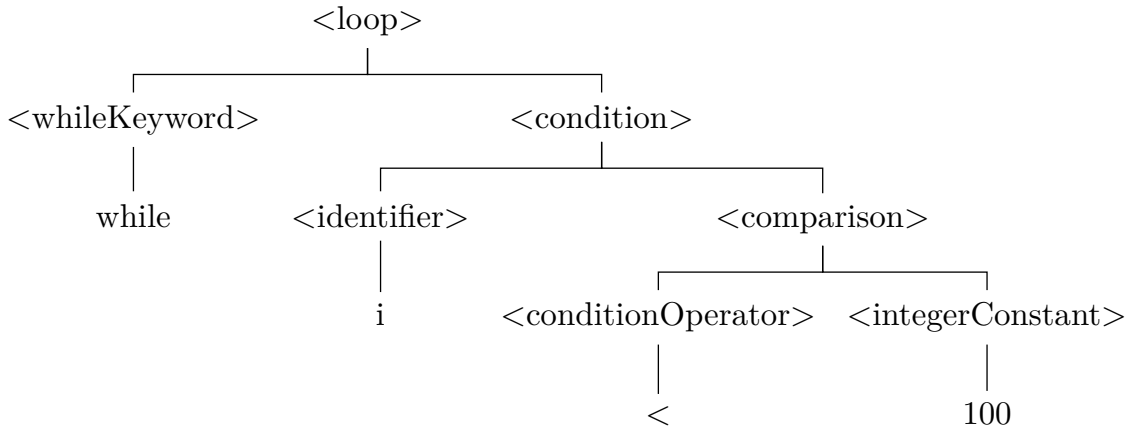



**Figure 2.3:** Example syntrax tree for the while-loop example.

### Semantic Analysis

The final step during the analysis phase is the semantic analysis. This step is performed in order to detect context-specific information of the syntax patterns found in the AST, mainly type checking, resolution of scope, declarations, as well as resolution of function overloading. One example of this process is given by the implicit conversion of an *Integer* value to a *Double* value necessary for a given operator.

Additionally, first optimizations of the code are applied to generate an intermediate representation for later use in the code generation process. These optimizations are machine independent and focused around improvements of the source code. Among others, these optimizations include the removal of redundant code, loop unrolling, function inlining, constant folding and constant propagation, as well as branch removal in a modern compiler.

#### Code Generation

The final part of the compilation process is the code generation step. Starting with the optimized intermediate representation, that is emitted by the code analysis phase; the statements are replaced with equivalent instructions of the target language. Incorporated into this step are machine dependant optimizations. This includes optimization due to knowledge of target architecture like cache sizes and parallelism capabilities.

## 2.4 Athariac: Code Transformation Framework

During every compilation process, the input source undergoes a variety of different representations within the compiler, starting with the CST, AST and finally, the compiled program. The transition between these different states are executed by *Transformations*. In the following, Athariac, the code transformation framework that is used in the ExaStencils code generator will be described briefly.

#### Introduction to Athariac

Athariac is a flexible code transformation framework that is heavily used within ExaStencils, as it provides the basis for the source-to-source compilation of ExaStencils' code generation. The underlying principle of Athariacs code transformation approach is stepwise term rewriting, altering the source to the target language by a large number of small transformations.

Athariac is implemented in Scala, an object-functional language with beneficial features for the creation of an external DSL. One advantage of using Scala is its *Parser Combinators*. They allow for the creation of a sophisticated parser by combining more straightforward implementations. In combination with Scala's syntax for the specification of language rules, which resembles the extended BNF notation, this facilitates the implementation of complex parsing functionality (2.15).

```
1  def expr: Parser[Any] = term~rep('+'~term |'-'~term)
2  def term: Parser[Any] = factor~rep('*'~factor | '/'~factor)
3  def factor: Parser[Any] = floatingPointNumber | '('~expr~')'
```

**Listing 2.15:** Scala parser combinators; modified according to Odersky.[7]

In accordance with the previous section, Athariac's parsing process emits a syntax tree and introduces the *Node* class as a base data type. All of Athariac's data structures inherit from *Node* and are implemented as a Scala *case class*. The usage of case classes enables Scala's pattern matching that fulfils an important role in Athariac's transformation process.

### 2.4.1 Transformation in Context of ExaStencils

The code transformation process in Athariac is conceptualized as a series of small changes to the code representation. These changes, *Transformation* traverse the AST and use pattern matching to identify specific nodes or subtrees. Once a successful match occurs, the

*Transformation* executed on the given node by either modifying it, deleting it from the AST or replacing it by one or more new nodes. A particular powerful characteristic of pattern matching in the context of *Transformation* is the possibility to match for specific member values in case class objects. Using this *deep matching* feature allows for easy identification of entire subtrees.

```
1   exp match{
2     case myExp@BinaryExp(_,_,DoubleConstant(3.14))
3          => println('Found␣PI␣' + myExp)
4     case myExp@BinaryExp(_,DoubleConstant(3.14),_)
5          =>  None
6     case _ => println('No␣PI')
7   }
```

**Listing 2.16:** Scala's deep matching functionality. Implementation example to match a particular binary expression. Modified according to Schmitt et al. [4]

Listing 2.16 demonstrates an example of the deep matching functionality. Implemented as a transformation, any binary expression with a right operand of 3.14 will be matched by this pattern, and the according code will be executed. In this example, a println statement will be evaluated. If the matched binary expression has 3.14 as the left operand, the corresponding node will be removed from the AST by returning Scala's None object. The transformation will then be applied recursively to the next node. However, since the recursive behaviour is not always desired, Athariac offers functionality to suppress the transformation from being applied to a match's subnodes.

### 2.4.2 Athariac Strategies

The transformation process that is implemented in Athariac demands a multitude of different *Transformations*, a not neglectable subset of which needs to be applied in a particular order. For example, *Transformations* that search for errors in the AST, or such that annotate certain information to the nodes must be applied before follow-up *Transformations* can be executed. On that account, to simplify the transformation handling and reduce the code maintenance cost, Athariac bundles transformation into *Strategies*.

Athariac comes with a *DefaultStrategy* class, that is designed as a simple container for consecutively applying *Transformations* from a list. Besides, a *CustomStrategy* base class is implemented in Athariac for more sophisticated transformation application. These allow for loops and branching of transformation execution as is necessary in a simplification strategy, that evaluates constant expressions during the compilation process (2.17).

```
1   object GeneralSimplify extends DefaultStrategy("Simplify␣
      expressions") {
2     var compactAST : Boolean = false
3
4     def doUntilDone(node : Option[Node] = None) = {
5       do {
6         apply(node)
7       } while (results.matches > 0)
8     }
9
10    this += new Transformation("Simplify", {
11      case add : IR_Addition => // handle Transformation
12      case sub : IR_Subtraction => // handle Transformation
13      ...
```

**Listing 2.17:** Model simplification strategy

The second task of strategies besides guidance of the transformation execution is to log information and transformation progress, as well as statistics for the number of matched nodes. This is done via communication with the *StateManager*.

### 2.4.3 StateManager

The *StateManager* handles the whole transformation process as it controls the strategies execution. For every transformation that is to be applied in a given strategy, the *StateManager* checks the AST for eligible nodes, such of type *Node* or *Collection[Node]*. Additionally, the *StateManager* provides *find()* and *findAll()* to identify nodes with certain attributes and apply transformation regionally.

The second task of the *StateManager* is to oversee the transformation process. A strategy that is to be applied opens a transaction with the *StateManager* receiving a transaction token in return. This ensures that only one transaction at a given time is issued and prevents the modification of intermediate code states between strategies which might lead to unwanted effects in the final output code. After a transaction is registered, the transformations of the strategy are executed, and statistics returned. Finally, if the transformations were successful, the changes are committed, and the transaction is closed. In case of an error, the *StateManager* can revert the AST to a former state or abort the strategies execution at all.

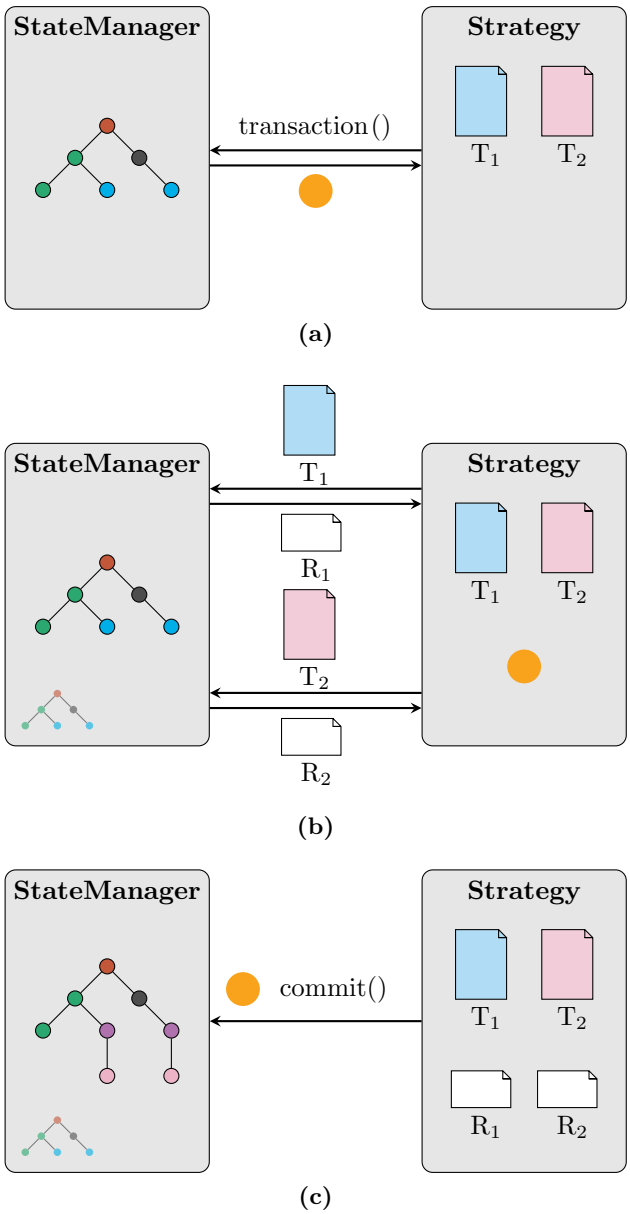



**Figure 2.4:** This figure shows the interaction between Strategies and the StateManager: a) A Strategy opens a new transaction with the StateManager and receives the transaction token, ensuring no concurrent transformation is executed. b) The Strategy applies its transformation sequence to the AST. Results and statistics are logged by the StateManager and returned to the Strategy. c) As the Strategy finishes, the transformations are committed to the AST and the transaction token is returned to the StateManger, closing the session.[3]

# 3 Motivation

ExaStencils already provides a wealth of transformation *Strategies* developed by different contributors. Besides, new *Strategies* will be introduced in the future. Unfortunately, with every developer having an individual coding style, and due to the number of existing *Strategies*, it is not reasonable for a single developer to know the functionality and implementation details of the latter. However, since all *Strategies* are applied to the AST and modify it, there is an influence of a *Strategy's* execution on all following *Strategies* leading to intrinsic dependencies between transformations. And though the *Strategies* are loosely clustered based on their functionality, their order was initially determined by a trial-and-error approach. Therefore an automation of *Strategy* arrangement paired with an examination of the *Strategies'* dependencies yields promissing potential to reduce code maintenance effort, as well as possible optimizations of the compilation process.

### Objective

The objective of this thesis is to provide an interface for automated *Strategy* positioning and gather dependency information between different *Strategies* in order to unify the various *Strategy* implementations and reveal possible optimization approaches.

### Requirements

The requirements for this thesis are split between the scheduling of transformation *Strategies* and the analysis of inherent dependencies. The former demands to unify the *Strategy* management which is initially given by a multitude of different implementations in coding styles, as well as in conception. Therefore, an interface must be developed and existing *Strategies* altered to fit the introduced handling approach. Furthermore, the hierarchical structure must be identified for the implementation of an efficient scheduling approach.

Leading to the second task of this thesis: the search space for finding intrinsic dependencies is overwhelmingly large, and to this end, coupled with non-neglectable computation time for every test case. Thus, a heuristic is inevitable to narrow the search space and allow for knowledge gain within reasonable time frames. Finally, a method is needed in order to resolve the hierarchical structure of the *Strategy* collection.

# 4 Automation of Strategy Positioning

In order to manage the transformation process ExaStencils implements a *LayerHandler* class for each of the four layers corresponding to ExaSlang I-IV, as well as for the intermediate representation and the final code generation process. So far, the strategy configuration is hard-coded in the according *LayerHandler* and their execution demands a variety of different setups and calls as illustrated in Listing 4. Furthermore, the amount of transformation strategies that need to be applied exceeds one hundred, complicating the introduction of new transformation strategies in the future. Apart from a simplification for a new developer to add transformation strategies, automation of strategy arrangement may be beneficial to implement problem specific optimization targets in the future, as it allows for flexibility in the strategy sequence.

```
1   1. ExampleStrat.apply()
2   2. while(condition){
3          ExampleStrat.applyAndCountMatches()
4      }
5   3. if(condition){
6          ExampleStrat.apply()
7      }
8      else {
9          AnotherStrat.doUntilDone()
10     }
11  4. ExampleStrat.get.convertToFunctions()
```

**Listing 4.1:** Different execution patterns for transformation strategies, examplified.

A first step towards the automation of strategy arrangement is to implement a unifying interface for new strategies and adapt already existing ones. Therefore, the *Schedulable* trait is introduced, demanding a apply()- and a reset()-method to be defined for inheritting strategies.

```
1   trait Schedulable {
2     def apply(applyAtNode : Option[Node] = None): Unit
3     def reset(): Unit = {}
4 }
```

**Listing 4.2:** The proposed Schedulable interface for strategies endorses the strategy.apply execution pattern.

Using *Schedulable* to adjust existing *Strategies* and create wrapper for more sophisticated strategy execution, prepares for the implementation of the Scheduler.

## 4.1 Strategy Scheduling

The idea for the *Scheduler* is to register *Strategies* in combination with yet unknown information about the *Strategies* requirements. Once all *Strategies* for a given layer and/or a specific problem are registered, the Scheduler sorts them according to their dependencies and executes the transformation *Strategies* sequentially.

### 4.1.1 Conception

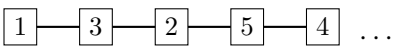
For the scheduling, the strategies and their inherit dependencies can be seen as a directed acyclic graph (DAG), with the directed edge between two vertices $v1$ and $v2$ as a dependency of *Strategy 2* on *Strategy 1*.

The problem of scheduling the *Strategies* then reduces to finding a sorted list of vertices such that for every directed edge $(v_i, v_j)$ between two vertices $v_i$ and $v_j$ the sorting index $i < j$. The resulting sequence of vertices then is called a topological sorting. It is easily proven, that there exists at least one topological sorting for every directed acyclic graph.

*Proof.* Let $G$ be a directed acyclic graph (DAG). Then follows that there must exist a vertex without incoming edges. Let $v_1$ be a vertex without incoming edges in $G$ . Then $G_1 = G - \{v_1\}$ is also a DAG. Now let $v_2$ be a vertex without incoming edges in $G_1$ . It follows that $G_2 = G_1 - \{v_2\} = G - \{v_1, v_2\}$ is a DAG. Finally, $G_0 = G_{n-1} - \{v_n\} = G - \{v_1, v_2, \ldots, v_{n-1}, v_n\}$ then the sequence $\{v_1, \ldots, v_n\}$ is a topological sorting since for every edge $(v_i, v_j)$ holds $i < j$. □

From this proof, an efficient algorithm for finding a topological order can be derived as is demonstrated in Figure 4.1.

Assuming the DAG in Figure 4.1a as a representation of *Strategies 1* to *9* with their dependencies as directed edges, e.g. *Strategy 3* is dependent on the prior execution of *Strategy 1*. In this example, *Strategy 1* and *Strategy 2* can act as possible starting points for the topological sorting. Figure 4.1b showcases the situation after *Strategy 1* has been removed from the graph. *Strategy 2* and *Strategy 3* are the next eligible candidates for the sorting. Adding *Strategy 3* to the sorted list yields the DAG of Figure 4.1c with only *Strategy 2* without requirements. Continuing in this manner leads to a final sorting as of Figure 4.1f with *Strategy 6*, *Strategy 8*, and *Strategy 9* as elgible candidates for the next step:




As proven previously, there exists at least one topological sorting for any DAG, but this sorting must not be unique, as there might be multiple eligible candidates in each step to choose from. Listing 4.3 shows a pseudo code variant of the demonstrated approach.

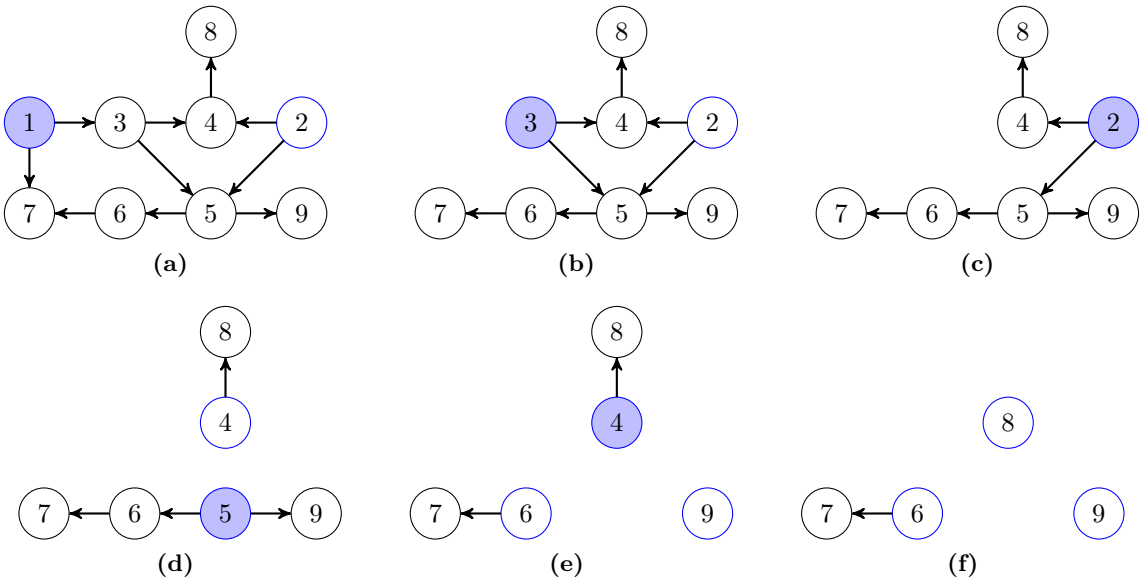



**Figure 4.1:** This figure demonstrates the topological sorting algorithm for an example DAG. In a) verteces 1 and 2 are eligible starting points for the sorting, since both have no incoming edges. Choosing vertex 1, removing it and its outcoming edges leads to the modified graph in b). Again, multiple verteces are suitable. Deciding for vertex 3 leads to the representation in c) with only the vertex 2 being qualified for removal. This process is continued until every vertex from the graph has been transferred to the sorted list. In the representations of d), e), and f) all eligible verteces are blue-rimmed, while the chosen vertex is additionally highlighted.

```
1            topologicalSort(Graph G)
2              if(G isEmpty)
3                return result;
4              else
5                find vertex x with incomingEdges(x) == 0;
6                result.add(x);
7                topologicalSort(G-x);
```

**Listing 4.3:** Pseudo code representation of a topological sorting algortihm. The DAG is searched for a vertex without incoming edges. Said vertex is added to the sorting and removed from the graph alongside its outcoming edges. The process is continued with the modified graph until the sorting is complete.

### 4.1.2 Implementation

In ExaStencils, the directed acyclic graph is stored in form of a *scala.mutable.Map* with the key-value pairs being the *Strategy* and a list of its requirements.

During the registration of a *Strategy* for the scheduling, an entry with its requirements is added to the strategy map.

```
1  strategyMap += (Strategy, requirementList)
```

In the following, since the Scheduler is designed to allow for dependency information being specified in both directions, the registered *Strategy* is added to the list of requirements for every *Strategy* being specified as dependent on the former.

```
1    for(entry <- dependentList){
2      strategyMap(entry) += Strategy
3    }
```

The scheduling itself (cf. Listing 4.4) is implemented according to the recursive pseudo-code example in Listing 4.3. Lines five to nine correspond to the first step of the portrayed algorithm, as every key-value pair in the generated *strategyMap* is traversed and examined for an empty *requirementList*. In order to avoid artificial duplicates in the sorted arrangement, the *Strategy* is added to the *strategyList* if it is not yet contained. Lines ten to thirteen correspond to the removal of the processed vertex from the graph. Line seventeen starts the recursion call.

```
1      override def scheduleStrategies(): Unit = {
2        if(strategyMap.size == strategyList.size){
3          return
4        }
5        for((k,v) <- strategyMap){
6          if (strategyMap(k).isEmpty){
7            if(!strategyList.contains(k)){
8              strategyList += k
9              }
10             for ((i,j) <- strategyMap){
11               if(strategyMap(i).contains(k)){
12                 strategyMap(i) -= k
13               }
14             }
15           }
16       }
17       scheduleStrategies()
18     }
```

**Listing 4.4:** Topological sorting as it is implemented in ExaStencils *Scheduler*. Lines 5-9 correspond to finding a vertex without incoming edges and saving it to the ordered list. Lines 10-13 remove the chosen vertex from the DAG. The recursive call in line 17 starts the next search iteration with the modified DAG.

## 4.2 Dependency Analysis

After the construction of a *Scheduler*, the focus of the second part of the thesis shifted towards the generation of dependency knowledge between the existing *Strategies*. With the *Strategies* being plentiful and their contents not necessarily being known by most of the developers, they resemble a black box in functionality. This results in an analytical approach to find sufficient dependency information between *Strategies* being deemed unsuitable.

### 4.2.1 Conception

The general idea behind extracting dependencies is to swap *Strategies* and examine the compilation process. If such a swap leads to the failure of the compilation process, an inherent dependency has been violated and can be deduced by comparison of the faulty configuration to successful ones. The main problem with this approach is the total amount of compilations needed in order to cover all possible arrangements. Layer I, for example, features 15 *Strategies* resulting in $1.3 \times 10^{12}$ possible configurations. With the generous assumption every compilation finishes in one second, the time needed roughly rounds to $15 \times 10^6$ days or 630.000 years. Since the total amount of individual *Strategy* sequences depends on the factorial of the *Strategy* count, which exceeds 100 in higher layers, it is clear that this approach is beyond computational capabilities. Another challenge is the extraction of dependency knowledge with limited information: The sequence of *Strategies* involved, as well as the success of the compilation attempt.

Concerning these difficulties, an iterative method is chosen for this thesis.

### 4.2.2 Creating Permutation

The first aim of the iterative approach is to reduce the search-space by introducing a permutation neighbourhood and only generate permutations locally. This makes sense in a way, that the *Strategies* are already loosely clustered according to the role they fulfil in the compilation process. A transformation strategy that is connected to the optimization step, for example, loop unrolling, does not need to be compared to a Strategy of the early semantic analysis, e.g. scope resolution. Dependencies of *Strategies* connected to the same compilation phase, that is in proximity within the *Strategy* sequence, are more significant to resolve. For that reason, the permutation neighbourhood size is set to two in the first iteration, hence only pairwise permutations are considered.

Figure 4.2 demonstrates the permutation process. Two directly neighbouring *Strategies* are sequentially swapped while the remaining *Strategies* are positioned according to the default configuration. For each of the resulting *Strategy* arrangements the compilation is later executed. This approach greatly reduces the number of necessary runs from 8! = 40320 to 8 for the shown example, albeit offering limited knowledge gain: Dependencies between *Strategies* in immediate proximity at best.

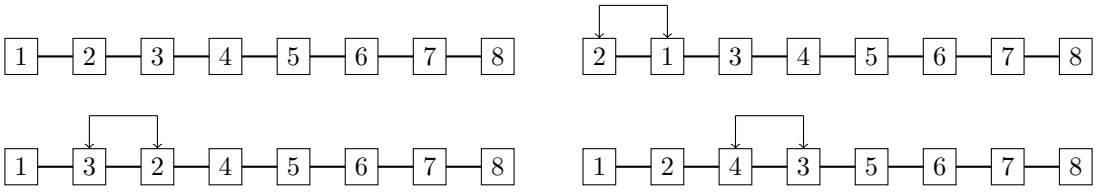



**Figure 4.2:** In this figure, the consecutive pairwise permutation approach is displayed for the first iteration with neighbourhood size set to two. The neighbourhood is traversed along the strategie sequence swapping the included strategies, while the remaining ones are left untouched.

Therefore, the following steps aim to increase the resolution with which dependencies can be detected by defining permutation neighbourhoods of larger sizes. Within such a neighbourhood, all possible permutations are created for each of which a *Strategy* configuration is generated with *Strategies* outside of the neighbourhood positioned according to the default. Once this process is finished, the neighbourhood is propagated along the *Strategy* sequence and the permutation process begins anew. The number of *Strategies* that the neighbourhood moves onwards is represented by the *overlap* value.

Figure 4.3 illustrates the functionality of the neighbourhood size and overlap. As can be seen, increasing the neighbourhood size exercises a great influence on the number of individual configurations that are generated and need to be run. Changing the neighbourhood size to three (4.3a) and setting the overlap (4.3b) to two results in 36 different *Strategy* sequences. Further increasing the neighbourhood size to four while deploying the same propagation method, advancing one Strategy per iteration, raises the number of Strategy *arrangements* to 120 (4.3d). However, some of the computational cost that comes with an increase in neighbourhood size can be mitigated by reducing the overlap between two following neighbourhoods. By changing the overlap to one, which results in advancing two *Strategies* at a given time in case of a neighbourhood size of three, the number of configurations is reduced to 20 (4.3c). This trade-off plays an important role in balancing the dependency resolution with reasonable computation times.

### 4.2.3 Dependency Extraction

The second aim of the iterative approach is to reduce the complexity of extracting dependency information from the generated data. In order to demonstrate, let 4-5-6-7 be a subsequence of *Strategies*. Using a neighbourhood size of four leads to the following configurations tagged with their compilation success:

| | | | |
|---|---|---|---|
| $4-5-6-7$ ✓ | $5-4-6-7$ ✓ | $6-4-5-7$ $X$ | $7-4-5-6$ $X$ |
| $4-5-7-6$ $X$ | $5-4-7-6$ $X$ | $6-4-7-5$ $X$ | $7-4-6-5$ $X$ |
| $4-6-5-7$ ✓ | $5-6-4-7$ $X$ | $6-5-4-7$ $X$ | $7-5-4-6$ $X$ |
| $4-6-7-5$ ✓ | $5-6-7-4$ $X$ | $6-5-7-4$ $X$ | $7-5-6-4$ $X$ |
| $4-7-5-6$ $X$ | $5-7-4-6$ $X$ | $6-7-4-5$ $X$ | $7-6-4-5$ $X$ |
| $4-7-6-5$ $X$ | $5-7-6-4$ $X$ | $6-7-5-4$ $X$ | $7-6-5-4$ $X$ |

The first step in resolving the dependencies is to focus on configurations with a permutation count of two, that is two of the *Strategies* are altered compared to the default. If the compilation process results in a failure, a dependency between the two swapped *Strategies* is derived. In the given example, this approach leads to $4 \rightarrow 6$ (Strategy 4 is a requirement for Strategy 6), $4 \rightarrow 7$, $5 \rightarrow 7$, and $6 \rightarrow 7$. However, if consequently applied, this approach leads to an overestimation of dependencies.

This behaviour is due to some dependencies being falsely identified, while others being redundant. The dependencies in the example above can be reduced to 4→6 and 6→7.

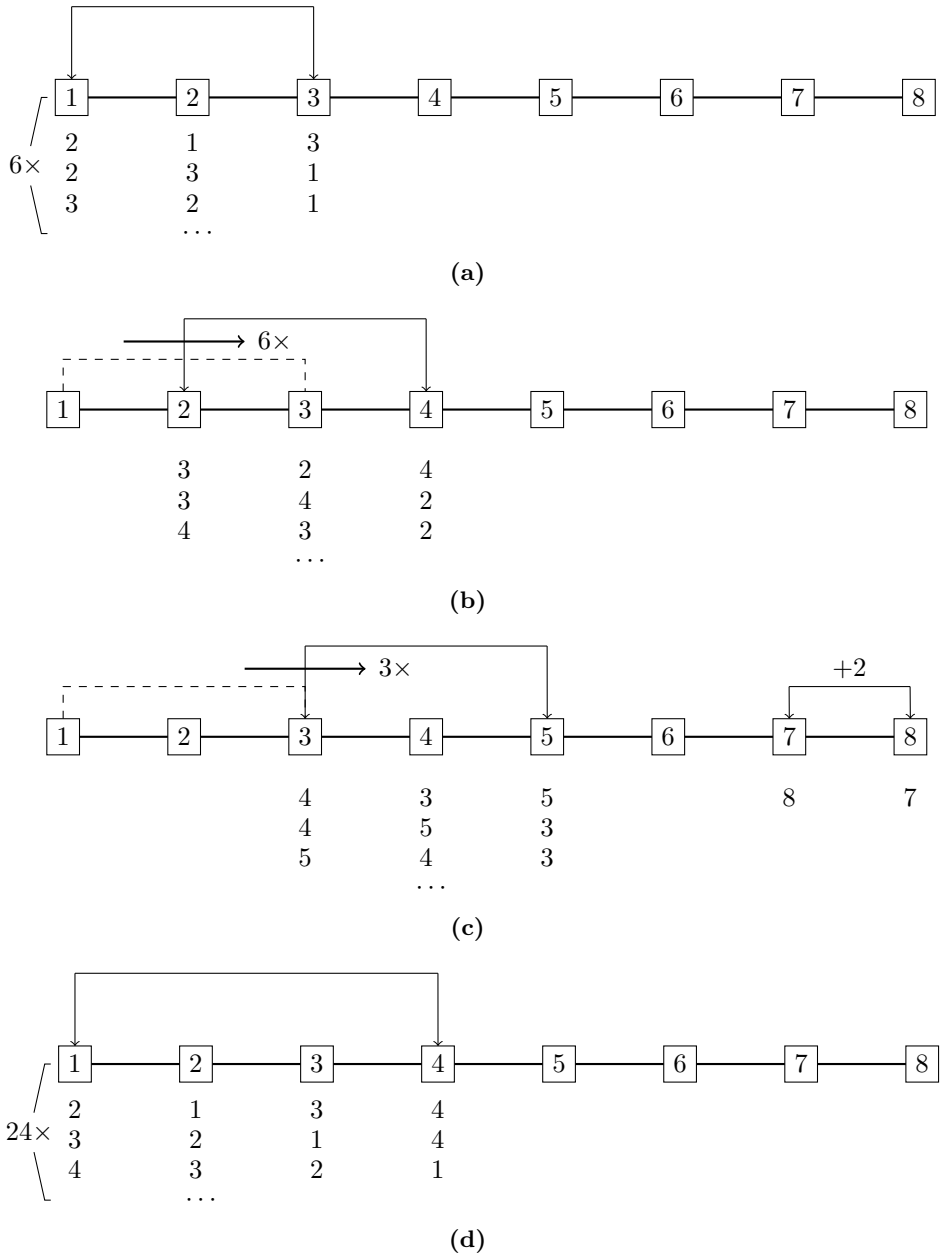



**Figure 4.3:** This figure portrays the concept of the neighbourhood size and the overlap along with the influence they exert on the total number of permutations. Having a neighbourhood size of 3 in a) leads to 6 possible arrangements within that neighbourhood. Combined with an overlap of 2, c.f. b), results in 36 different permutations, as the neighbourhood size fits the sequence 6 times. Reducing the overlap to 1 however, reduces the total number of possible configurations to 20 as the neighbourhood travers the strategy sequence in 3 steps, leaving 2 additional permutations for the remaining two strategies. An increase in neighbourhood size as shown in d) has a great influence on the total number of permutations possible. An increment of the neighbourhood size from 3 to 4 leads to 24 different possibilities instead of 6 prior.

By iteratively increasing the neighbourhood size and varying the overlap, dependencies are detected according to the *Strategies*' proximity. Following the first iteration, swapping of

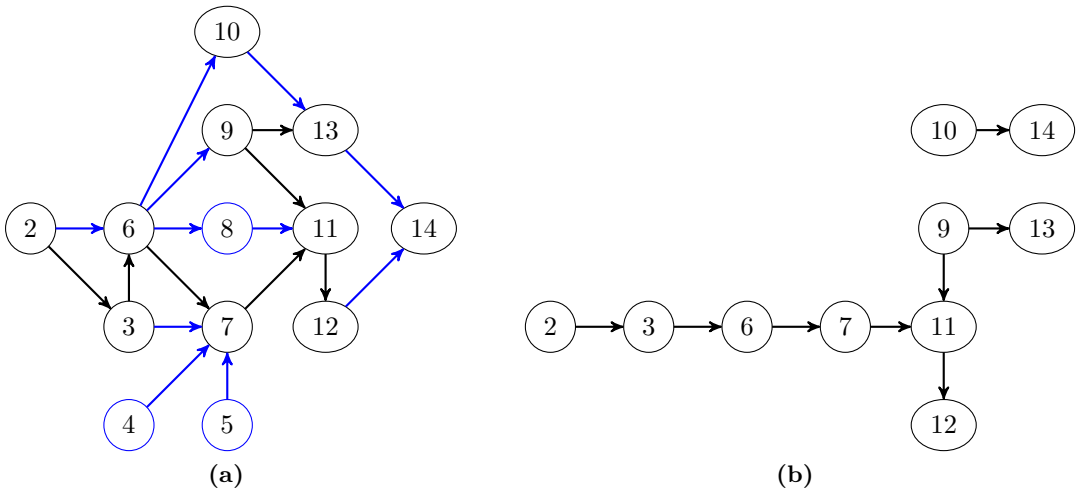



**Figure 4.4:** This figure depicts the effects of overestimating dependencies. a) shows all dependencies that are discovered. However, some relations, highlighted in blue, are misleading as they violated separate dependencies. b) shows the adjusted dependency graph using the iterative approach increasing the permutation neighbourhood.

directly neighbouring *Strategies*, a list of reliable dependency information is obtained, that is used to filter misleading dependencies. After each run, the dependency information is updated for use in the upcoming iteration.

### 4.2.4 Implementation

Regarding the implementation, some changes are made to the *Scheduler* to extend its utility. First, a registerList()-method is implemented that allows for the registration of an already sorted *strategyList*. This method facilitates changing between several permuted configurations. Besides, a *Permutation* object is introduced for each *Layer* that provides the *Layer's* default configuration as a *strategyList* and acts as the connection between the permutations and the *Scheduler*.

#### Creating Permutations

The generation of the *Strategy* permutations is a combination of ExaStencils' *main* object, the layer-specific *Permutation* object, as well as the layer-specific *LayerHandler* and the *DefaultScheduler*. In order to fulfil the tasks needed, the *main* is extended by a set of five layer-specific permutate methods (L1_permutate - IR_permutate) and the entry point for the code generation process is adapted to fit the iterative nature of the dependency data acquisition approach. Structurally, the *main* receives the default configuration of the examined *Layer* provided by the congruous *Permutation* object. The permutation parameters, neighbourhood size and overlap, are passed to *Permutate* and create an interval of *Strategies* in the default configuration to be permuted. Unaffected *Strategies* are discarded at

this point. Every permutation within the defined neighbourhood is created and for each individual *Strategy* configuration snippet, the code generation process is initialized.

The compilation begins with processing the command line arguments. Hence, the ExaSlang source and the problem specifics are prepared. Hereafter, a new *Strategy* configuration for the examined Layer is constructed using the aforementioned configuration snippet. Therefore, every element from the default configuration up to the lower bound of the permutation neighbourhood is added to the new *Strategy* list, followed by the *Strategies* from the permutation. The list is finalized by prepending the remaining Strategies from the default, and returned to *Permutation*. The compilation process continues with the initialization of the *LayerHandler* and executes its *handle()* method, which itself registers the permuted *Strategy* list with the *DefaultScheduler*. Finally, the *DefaultScheduler* applies the *Strategies* in sequence, and the compilation process for the examined layer is wrapped up. Previous and following layers are handled, if necessary, according to their default implementation.

Once a compilation for every permutation within the neighbourhood is executed, the permutation neighbourhood is propagated along the default *Strategy* sequence as dictated by its size and the overlap. Finally, after the permutation neighbourhood included every *Strategy* at least once, the neighbourhood size is incremented by one and the next iteration starts.

#### Dependency Extraction

The dependency analysis begins with the separation of the individual permutation runs and storing them according to the neighbourhood size that was used. The new *Configuration* class is introduced to store important data for every single run:

- Data from the compilation log:
  - compilation time
  - *Strategy* list
- data by comparison with the reference run:
  - compilation result, i.e. success or failure
  - permutation count
  - positional changes
- created during dependency resolution:
  - violatiod dependencies

While the *Configurations* are extracted, the compilation output associated with the run is compared line-by-line to the log of a reference compilation. If the comparison contains additional compiler warnings or compiler errors, the compilation process is deemed a failure and the *Configuration* is tagged as faulty.

In order to prepare the *Configurations* for the dependency analysis, the permutation count is calculated by counting the number of positional changes in the *Strategy* sequence compared to the reference. Simultaneously, the new indices of the altered *Strategies* are stored alongside, as they are used later to determine whether the *Configuration* violates any resolved dependencies.

The actual dependency extraction begins with the analysis of all *Configurations* generated with a permutation neighbourhood of size two. In the first step, the *Configuration* collection is filtered for faulty *Configurations*. Since the permutation neighbourhood allows for pairwise swapping of adjacent *Strategies* only, the dependencies are derived from the positional changes of the very same *Strategies*. All dependencies that are discovered in this manner are added to a *DependencyInfo: map(Strategy -> List(Requirements))*. In the next steps of the analysis, the *DependencyInfo* is used to update the violation status of all *Configurations* with a neighbourhood size larger than two, by comparing the index of a *Strategy* in the particular *Configuration* to its requirements. Filtering *Configuration* from the next higher neighbourhood size for a failure in compilation result, a permutation count of two and no violated dependencies allows for the extraction of additional dependency knowledge. The process of altering the neighbourhood size, resolving dependencies for applicable *Configurations*, updating the *DependencyInfo* and updating the violation status, is continued until all faulty *Configuration* are resolved by violating a known dependency. Otherwise, the process is stopped when the amount of unresolveable *Configurations*, i.e. *Configurations* with a permutation count larger than two, is constant for several iterations. Finally, any *Configuration* of the latter category is emitted to solve manually and the *DependencyInfo* is returned.

# 5 Evaluation

In the previous chapter, the conceptualization of the Strategy Scheduling and the Dependency Analysis were described, and their implementation details were explained. This chapter aims to evaluate the chosen approaches regarding the predefined requirements. Therefore, the integration of the Strategy Scheduler into the ExaStencils framework is examined. In a second step, the results of the Dependency Analysis are demonstrated exemplarily in the case of Layer I and the deployed heuristic is discussed.

## 5.1 Strategy Scheduling

In the first step of this thesis, the *Schedulable* trait was introduced to unify the *Strategy* handling. Using this interface endorses the *Strategy.apply* execution pattern that is predominantly used in the default LayerHandler approach. Therefore, the trait applies to the majority of ExaStencils' *Strategy* implementations with only minor changes needed. *Strategies* which diverge from the *Strategy.apply* pattern demand more implementation effort. The most common execution patterns besides the default are branching and looped repetition until certain criteria are met. The *Schedulable* interface allowed for the creation of generalized wrappers for these use cases, conforming to the *Strategy.apply* scheme. In addition, some transformational processes in the compilation resemble *Strategies*, alas are not implemented as a *Strategy* type. Particular Schedulable objects for each scenario were created and added and integrated into the unified Strategy handling approach.

At specific check-points in *L4_LayerHandler* and *IR_LayerHandler*, the *Strategy.apply* loop is interrupted to execute additional code statements: println, reset of data containers, or verification of a particular compilation stage. Though they could be modelled as a *Schedulable* object due to the distinct functionality they provide at particular time-points in the compilation process, these statements are treated as breakpoints in the *Strategy* execution. This results in multiple subsequences as *Strategies* up to such a breakpoint are registered, sorted and executed before the *Scheduler* is reset.

Overall, while the *Scheduler* increases code readability within the LayerHandler, the problem of *Strategy* positioning cannot be completely stripped from the developer at this point. However, functionality is provided to extend the *Scheduler's* utility considering the aforementioned cases in the future.

The introduced topological sorting is a reasonably efficient implementation of the scheduling process. However, it is to add that the development effort for future *Strategies* is not reduced to a large extend since inherent dependencies must be resolved beforehand. Finally, the *Scheduler* enables for a quick change of different *Strategy* combinations in consecutive compilations, a functionality that was initially not given in the ExaStencils implementation, and which might facilitate problem-specific optimizations in future work.

## 5.2 Dependency Analysis

The introduced method to identify intrinsic dependencies was applied to a list of example problems for each layer. In general, the dimension and problem class were varied, beginning with Layer II also the discretization method. Table 5.1 shows the employed problem setups for every layer, with the permutation parameters, neighbourhood size and overlap, given as tuples. All associated ExaSlang implementations were taken from the ExaStencils examples and can be found within the project's git repository[1].

| | Dimension | Discretization | Problemclass | Permutation |
|---|---|---|---|---|
| L1 | 1D, 2D, 3D | FD | Poisson, Stokes | (2,1) - (5,4) |
| L2 | 2D, 3D | FD, FV | Poisson, Stokes, Optical Flow(FD) | (2,1) - (5,4) |
| L3 | 2D | FD | Poisson | (2,1) - (5,1) |
| L4/IR | 2D, 3D | FD, FV | Poisson, Stokes, Optical Flow(FD) | (2,1) - (5,1) |

**Table 5.1:** Problems setups that were used during data generation for the dependency analysis. Permuation parameter are given as tuple of neighbourhood size and overlap.

Consistent with the representation introduced in Chapter 4 (c.f. Figure 4.1) the extracted dependencies are modelled as a DAG. The general dependency hierarchy resolved for Layer I is depicted as such a DAG in Figure 5.1 showing all dependencies that have been derived in any of the experimental setups. Though the different setups showed minor differences, the majority of the dependencies is constant throughout the different problems and does not allow for a generalization without further research. However, the analysis for Layer II showed distinct dependencies between *UnfoldKnowledgeDeclarations* and *ResolveRelativeLevels*, as well as *ResolveRelativeLevels* and *InlineDeclaredExpressions*. This observation indicates the problem-specific nature of some dependencies.

While the proposed method could resolve a presumably large subset of the dependency structure for Layer I, representing 11 of 15 strategies in total, its significance is reduced in higher layers. This is mainly due to two reason:

(a) As the total number of strategies in higher layers increases, the fraction of strategies that can be captured within a given permutation neighbourhood decreases, thus negatively affecting the dependency resolution.

(b) Since higher ExaSlang layers reduce abstraction, problem specifications tend to grow, increasing the necessary parsing effort and the AST size. Coupled with a larger amount of strategies that are to be applied, the compilation complexity is raised.

In addition, larger strategy sequences demand more iterations with a particular neighbourhood size. In conclusion, the reduced resolution combined with a severe increase in compile time leads to fewer dependencies resolved, covering a smaller fraction of the entire dependency hierarchy.

[1] https://i10git.cs.fau.de/exastencils/release

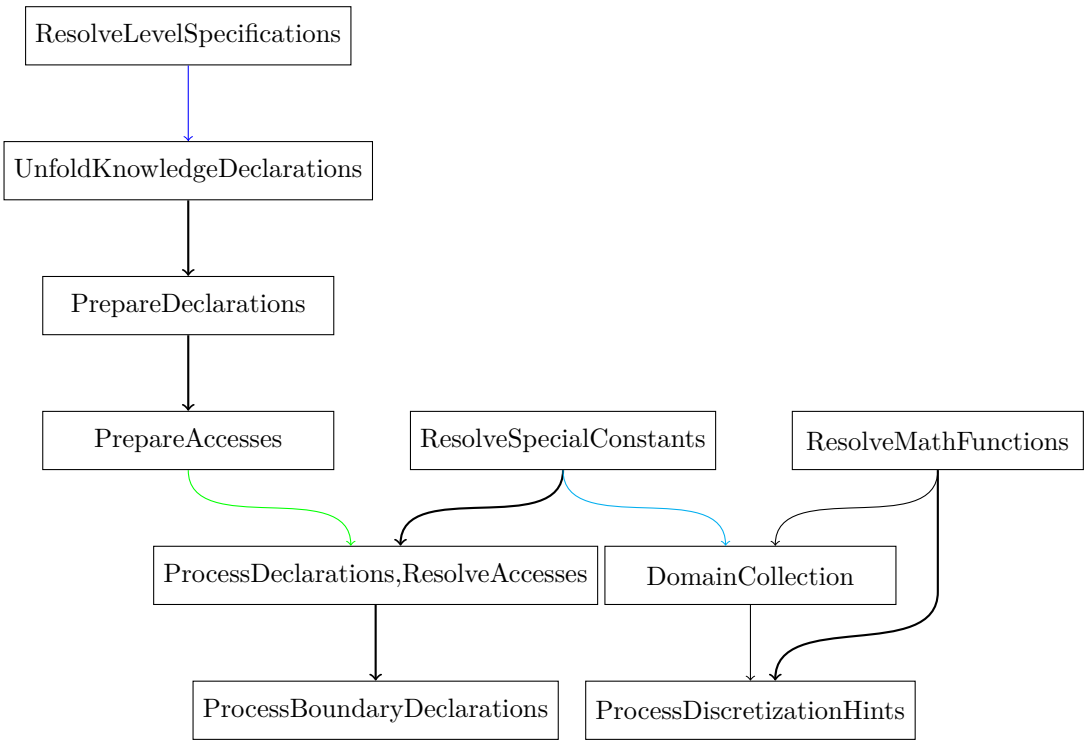



**Figure 5.1:** The dependency graph extracted for Layer I using the aforementioned problem specifications. The constant backbone of the dependency structure is tagged with bold arrows, the blue dependency only occurs for the 1D/3D FD Poisson and 1D FD Stokes problems, the green tagged arrow corresponds to 1D and 3D FD Poisson while the cyan labeled dependency only occured in case of 1D FD Poisson.

Furthermore, a linear rise in compile time for consecutive compilations was observed during this thesis, as shown in Figure 5.2. Albeit this behaviour occurs in every layer, it is discerned to be more severe for higher ones. Since each *Strategy's* contribution to the total runtime constant, and the size of the AST appears untouched, this phenomenon is partly contributed to an increase in transformation execution time. This lead to the implementation of reset functionalities for the StateManager and *Schedulable* objects, which mitigate the effects to an extent. However, this behaviour is not yet fully resolved, and further investigation is necessary.

In summary, the proposed heuristic is suitable for lower layers with few *Strategies* and faster compilation times. However, the approach suffers from the huge impact, an increase in resolution, i.e. increase in permutation neighbourhood size, has on the compile time. Therefore the introduced method can serve as a basis for the extraction of large dependencies structures and resolve dependencies between *Strategies* in proximity, nevertheless, additional work is necessary to derive the entire hierarchy.

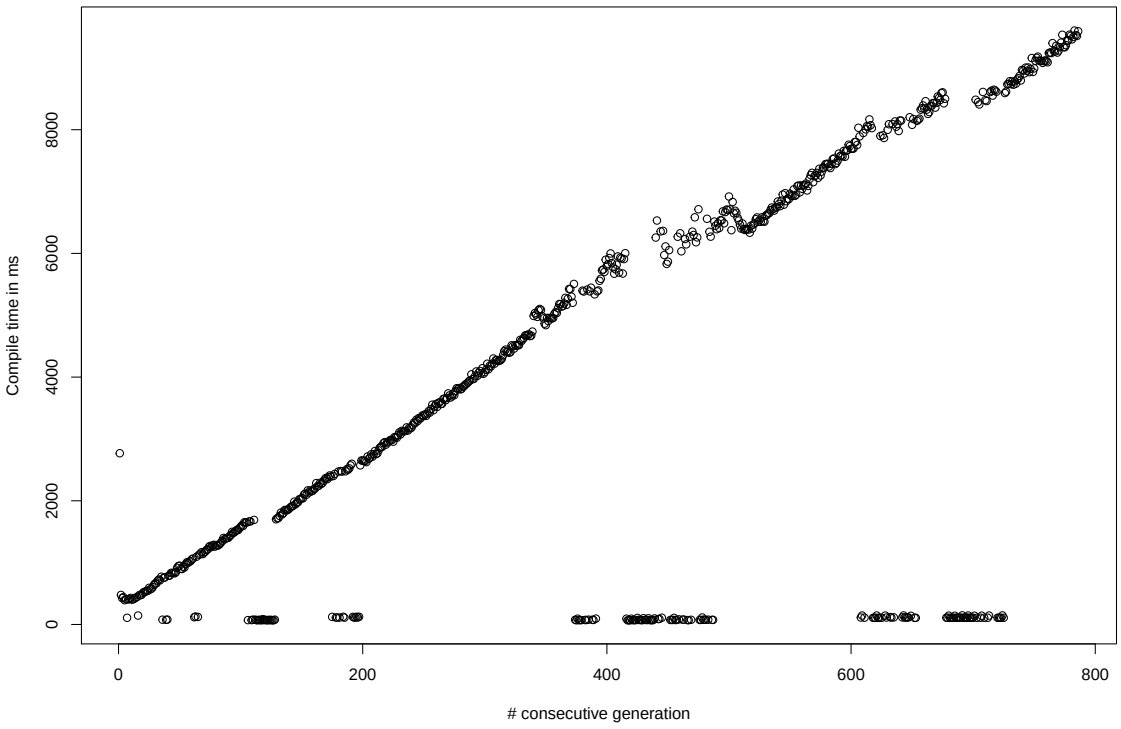



**Figure 5.2:** Linear increase in runtime as it was observed for consecutive compilations, i.e. employing the next strategy sequence. Shown for Layer III with the 2D FD Poisson example.

# 6 Conclusion

Within the scope of this thesis, a concept for unified transformation strategy handling was proposed, consisting of the Schedulable interface and the DefaultScheduler strategy execution manager. Utilizing the Schedulable trait, all existing strategies were altered to fit the unified approach; including strategies whose execution pattern diverge from the previous conception.

In addition, the DefaultScheduler promises to reduce development effort for the implementation of future strategies by replacing the need for manual strategy arranging with an automated strategy positioning; provided sufficient information of the hierarchical structure is available. Furthermore, changing between different strategy sequences in consecutive compilations is greatly simplified, providing a basis to approach problem-specific optimization in future projects.

In the second part of the thesis, an approach to derive the inherent dependency structure within the strategy collection was proposed. By generating all permutations within a defined interval on the strategy sequence, dependency information within that neighbourhood could be resolved in reasonable computation time. The introduced approach showed promising results for the lower layers. With the increase of the total number of strategies, as well as the computational effort for the compilation in higher layers, the maximum neighbourhood size feasible is critically limited, thus reducing the resolution to detect dependencies. Nevertheless, the methods are suitable to resolve dependencies between strategies in proximity and extracted a major subset of the hierarchical structure for Layer I.

Finally, this thesis was able to reveal formerly unknown shortcomings of ExaStencils implementation, especially longevity of transformation results, and thus contribute to further improvements in the code generation framework.

Future projects might expand upon the results of this work by choosing a different heuristic and resolve additional dependencies around the basis that is provided within this thesis. Besides, different optimization targets can be approached, for example, by examination of a particular strategy's impact on the AST size, or further investigation of problem-specific dependencies.

# Bibliography


[1] C. Lengauer, S. Apel, M. Bolten, S. Chiba, U. Rüde, J. Teich, A. Größlinger, F. Hannig, H. Köstler, L. Claus, A. Grebhahn, S. Groth, S. Kronawitter, S. Kuckuk, H. Rittich, C. Schmitt, and J. Schmitt, "ExaStencils – Advanced Multigrid Solver Generation," in *Software for Exascale Computing – SPPEXA 2016-2019*, ser. Lecture Notes in Computer Science and Engineering, Springer, 2020. [Online]. Available: https://www12.cs.fau.de/downloads/hannig/publications/ExaStencils_Advanced_Multigrid_Solver_Generation.pdf.

[2] C. Lengauer, S. Apel, A. Größlinger, A. Grebhahn, S. Kronawitter, M. Bolten, H. Rittich, F. Hannig, H. Köstler, U. Rüde, J. Teich, S. Kuckuk, and C. Schmitt, "ExaStencils: Advanced Stencil-Code Engineering," in *Proceedings of Euro-Par 2014: Parallel Processing Workshops*, (Porto), ser. Lecture Notes in Computer Science (LNCS), UnivIS-Import:2015-04-16:Pub.2014.tech.IMMD.inform.exaste, vol. 8806, Berlin; Heidelberg: Springer-Verlag, Aug. 25–26, 2014, pp. 553–564, ISBN: 978-3-319-14312-5. DOI: 10.1007/978-3-319-14313-2_47. [Online]. Available: http://link.springer.com/content/pdf/10.1007/978-3-319-14313-2_47.pdf.

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

[4] C. Schmitt, S. Kronawitter, F. Hannig, J. Teich, and C. Lengauer, "Automating the Development of High-Performance Multigrid Solvers," *Proceedings of the IEEE*, vol. 106, pp. 1969–1984, 2018. DOI: 10.1109/JPROC.2018.2854229.

[5] S. Kuckuk, "Automatic code generation for massively parallel applications in computational fluid dynamics," doctoralthesis, FAU University Press, 2019, xi, 243 Seiten. DOI: 10.25593/978-3-96147-274-1.

[6] D. Watson, *A Practical Approach to Compiler Construction*. Springer International Publishing, 2017, ISBN: 978-3-319-52787-1. DOI: 10.1007/978-3-319-52789-5.

[7] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*, 2nd. Sunnyvale, CA, USA: Artima Incorporation, 2011, ISBN: 0981531644.

[8] C. Schmitt, S. Kuckuk, H. Köstler, F. Hannig, and J. Teich, "An evaluation of domain-specific language technologies for code generation," in *2014 14th International Conference on Computational Science and Its Applications*, 2014, pp. 18–26.

[9] V. Turau and C. Weyer, *Algorithmische Graphentheorie*. Berlin, Boston: De Gruyter, 2015, ISBN: 978-3-11-041732-6. DOI: https://doi.org/10.1515/9783110417326. [Online]. Available: https://www.degruyter.com/view/title/510211.

[10] H. Köstler, C. Schmitt, S. Kuckuk, S. Kronawitter, F. Hannig, J. Teich, U. Rüde, and C. Lengauer, "A Scala Prototype to Generate Multigrid Solver Implementations for Different Problems and Target Multi-Core Platforms," *International Journal of Computational Science and Engineering*, vol. 14, pp. 150–163, 2017. DOI: 10.1504/IJCSE.2017.10003829.

[11] C. Schmitt, S. Kuckuk, H. Köstler, F. Hannig, and J. Teich, "An Evaluation of Domain-Specific Language Technologies for Code Generation," in *Proc. of the 14th International Conference on Computational Science and its Applications (ICCSA)*, (Minho, Guimaraes), UnivIS-Import:2015-04-16:Pub.2014.tech.IMMD.inform.aneval, New York, NY, USA: IEEE Press, Jun. 30–Jul. 3, 2014, pp. 18–26, ISBN: 978-1-4799-4264-0. DOI: 10.1109/ICCSA.2014.16.

# Apendix

```
Knowledge {
dimensionality = 2

minLevel       = 0
maxLevel       = 8
}

Ω = ( 0, 1 ) × ( 0, 1 )

f ∈ Ω = PI**2 * cos ( PI * x ) - 4.0 * PI**2 * sin ( 2.0 * PI * y )
u ∈ Ω = 0.0
u ∈ ∂ Ω = cos ( PI * x ) - sin ( 2.0 * PI * y )

op = - Δ

uEq: f = op * u

DiscretizationHints {
f on Node
u on Node

op on Ω

uEq

discr_type = "FiniteDifferences"
}

SolverHints {
generate solver for u in uEq

solver_targetResReduction = 1e-10
}

ApplicationHints {
l4_genDefaultApplication = true
}
```

**Listing 1:** Complete 2D Poisson specification for ExaSlang I

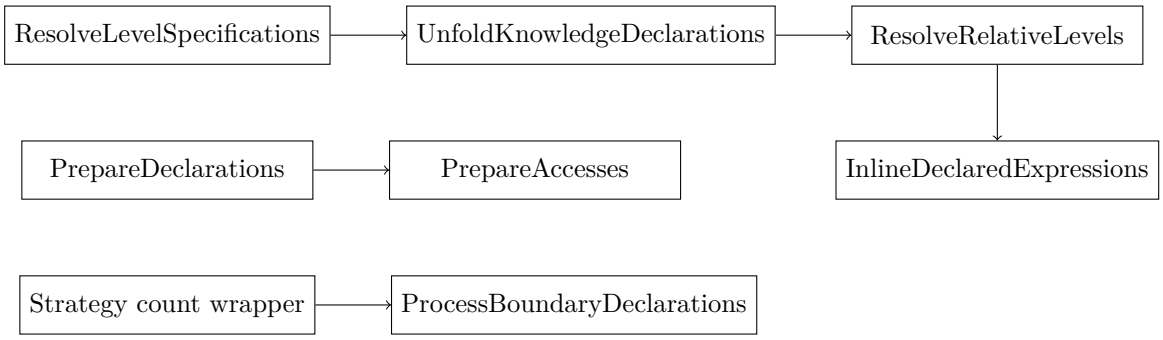



**Figure .1:** The dependency graph extracted for Layer II using the aforementioned problem specifications. The strategy count wrapper includes: ProcessDeclarations, ResolveAccesses, ResolveIntegrateOnGrid, and ResolveEvaluateOnGrid.

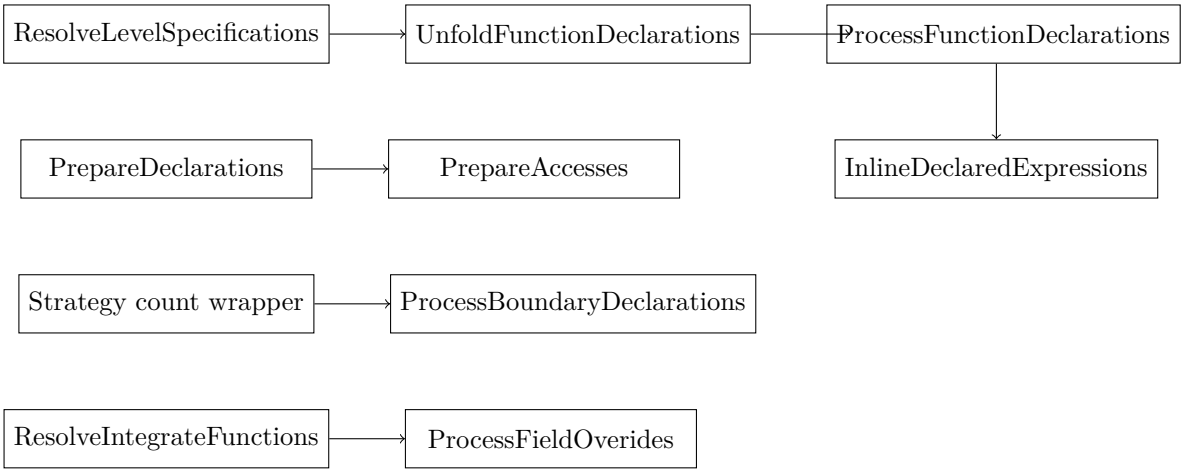



**Figure .2:** The dependency graph extracted for Layer III using the aforementioned problem specifications. The strategy count wrapper includes: ProcessDeclarations, ProcessSolverForEquations, ResolveAccesses, ResolveIntegrateOnGrid, and ResolveEvaluateOnGrid.

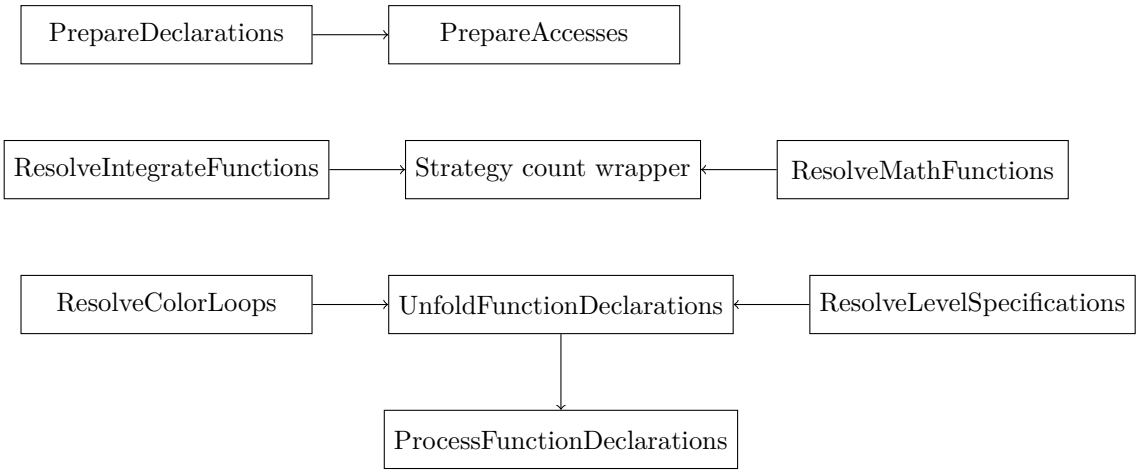



**Figure .3:** The dependency graph extracted for Layer IV using the aforementioned problem specifications. The strategy count wrapper includes: ProcessDeclarations, ResolveAccesses, ResolveIntegrateOnGrid, and ResolveEvaluateOnGrid.