University of Passau
Department of Computer Science and Mathematics

Bachelor's Thesis

Multigrid for the SPIRAL prototype in Scala

Sebastian Schweikl

September, 2017

Advisors: Prof. Lengauer, PhD*
Prof. Dr. Bolten**

* Department of Computer Science and Mathematics, University of Passau
** Department of Mathematics and Science, University of Wuppertal

Colophon (TEXnical details)

This thesis was typeset using LaTeX 2ε and the `scrbook` document class. The typographic style was heavily influenced by the KOMA-Script manual, which was written by Markus Kohm. For the construction of the page layout the `typearea` package was put to work with the option `DIV=9`. This way, the margins take up exactly one-ninth and two-ninths of the page in both width and height. Additionally, the type area and the page share the same proportions. Bitstream Charter, chosen at size of 11pt/14.5pt, served as main font for body text. This setup leads to an average amount of 75 characters per line and 38 lines per page, thus striking a good balance between legibility and economic efficiency. Other fonts used in this document include Source Sans Pro for headings and Luxi Mono for source code listings. Print version with binding correction and optimization of paragraph and page breaks.

I would like to thank my advisors Prof. Lengauer, PhD, and Prof. Dr. Bolten for their support and guidance throughout this thesis. I am also grateful for the help of Stefan Kronawitter, member of the ExaStencils project, who gave me valuable insights into the field of stencil computations. Last but not least, I gratefully acknowledge the help I received from Dr. Georg Ofenbeck who gave me answers on various questions I had regarding the SpiralS code generator.

Abstract

SpiralS is a partial reimplementation of the SPIRAL program synthesis system in Scala, covering the generation of fast numerical kernels for the discrete Fourier transform. The main goal of SpiralS, however, is not (only) the pursuit of performance but rather to propose a systematic and principled approach that allows for a more maintainable and re-usable construction of such generators. In this thesis, we follow these proposals and present a prototypical extension of SpiralS by the domain of multigrid methods. Specifically, our multigrid solver accepts a mathematical specification of a discretized square 2-dimensional Poisson equation with Dirichlet boundary conditions and outputs performance-optimized code in C. We focus on the implementation of the generator in particular and describe how it was realized. We craft different domain-specific languages (DSLs) to represent algorithmic and domain-specific knowledge on multiple levels of abstraction. The derivation of an appropriate solving algorithm for the given equation is driven by recursively rewriting DSL expressions. The application of performance optimizations is achieved by gradual refinement of the algorithm's structure. For this, we also discuss the translation between DSLs and we demonstrate how stencil codes can be used to improve both the implementation of the generator and the performance of parts of the generated code. In the end, we present and analyze the code output for various small example use cases.

Table of Contents

1 Introduction and Motivation 1

2 Background and Fundamentals 3
2.1 The SPIRAL Code Generator 3
2.1.1 Key Concepts 3
2.1.2 Multigrid Methods in SPIRAL 4
2.1.3 SPIRAL in Scala 8
2.2 Lightweight Modular Staging 9
2.2.1 Key Principles 9
2.2.2 Inner Workings 12
2.2.3 Important Utility Functions 14
2.3 Scala-Virtualized 15
2.3.1 External Functions 15
2.3.2 Language Virtualization 16
2.3.3 Reifying Static Information 17

3 Implementation 19
3.1 Overview of the Generator Architecture 19
3.2 Signal Processing Language 20
3.2.1 Matrices 20
3.2.2 Operators 21
3.3 Breakdown Rules 28
3.4 Applying Domain-Specific Optimizations 31
3.5 An Internal Representation of the Programming Language C 38
3.6 Translation from SPL to CIR 42
3.6.1 Atomic Expressions 45
3.6.2 Composite Expressions 47
3.6.3 Different Code Styles and Data Representations 50
3.7 Stencil Computations 52
3.7.1 Current Limitations 52
3.7.2 Fundamentals and Terminology 53
3.7.3 Toeplitz Matrices 54
3.7.4 Tensor Product of Toeplitz Matrices 57
3.8 Unparsing to C Code 61

4 Use Cases 65
4.1 Setup 65
4.2 Examples and Code Analysis 66

5 Conclusions 71

A Generated Code for a Multigrid Cycle 73

Bibliography 79

List of Acronyms 83

List of Figures

2.1 Expression trees as defined by `BaseExp` 13

3.1 Program generator architecture 20
3.2 SPL architecture 32
3.4 Depiction of a 2-dimensional stencil on a square grid 54

List of Tables

2.1 SPIRAL SPL breakdown rules for a multigrid solver 7

3.1 Mapping of SPL matrices to their appropriate SPL counterparts 22
3.2 Supported matrix operations in SPL programs 24
3.3 Summary of matrix operators and their appropriate IR nodes 27
3.4 Optimizing rewrites for SPL expressions 36
3.5 Translation of matrix formulas into code 43

Chapter 1

Introduction and Motivation

The efficient solution of discretized partial differential equations (PDEs) is a critical part of many applications in the field of engineering and scientific computing [1, p. 1, 2, p. 2]. This observation has led to a pursuit of creating ever faster solvers for PDEs. So called multigrid methods and stencil computations are widely used in this context but the process of implementing them is extremely time-consuming, especially when high performance and portability are desired [2, p. 2]. For this reason, automatic program generation systems such as the ExaStencils project[1] offer an appealing solution for this problem.

Encouraged by the success of earlier code generators, i.a. SPIRAL, the vision of ExaStencils is that domain experts need only provide a high-level mathematical description of the PDE to solve, augmented with hardware specifications of the target platform [2, p. 2]. The program generator then automatically synthesizes highly efficient code for the particular scenario at hand.

However, developing such generators is considered a difficult and daunting task due to the complexity of both the design and the actual implementation [3, p. 125]. First, it requires the designer to gain expertise in the domain in question, and second, code generators are compilers in their own right, which poses further challenges of efficient compiler design, such as understanding low level details of the target platform [4, p. 2].

To account for all this, ExaStencils follows a radically new approach: turning away from general-purpose languages and moving towards multiple specialized domain-specific languages (DSLs) results in great expressive power for a small domain of programs, i.e. stencil codes in this case, and refining programs gradually while exploiting the domain-specific knowledge available at each step ultimately enables high performance [5, pp. 554 sq.].

The success of program generators for high-performance code is measured in the performance of the generated code, but elegance in design, re-usability and maintainability of their implementation are often underrated [4, p. 2]. In all this

1 http://www.exastencils.org/

respects, SPIRAL is known for the highly efficient code it generates but it has also been opened up to the new domain of multigrid solvers. In this thesis, we try to achieve a similar extension for SpiralS, a partial reimplementation of SPIRAL in Scala.

Specifically, we build a small prototypical program generator focusing on a single kind of PDE only, namely discretized square 2-dimensional Poisson equations with Dirichlet boundary conditions. Thereby, we follow the principled approach taken by SPIRAL and ExaStencils while also being guided by SpiralS' implementation philosophy.

The rest of this thesis is structured as follows. The next chapter imparts the background knowledge and theoretical foundation necessary for the reader to fully comprehend the remainder of the thesis. This includes a brief overview of SPIRAL and SpiralS, the formulation of multigrid methods in SPIRAL, the multi-stage programming framework LMS and the closely related Scala-Virtualized compiler. Chapter 3 explains the implementation of the code generator in full detail. This encompasses an architectural overview, the construction of two internal DSLs, the translation between those languages and the application of performance optimizations. In this context, we also demonstrate how stencil computations can be used to improve both the code quality of the generator and the performance of the generated code. Chapter 4 presents various generator use cases, followed by an analysis of the corresponding fixed-size numerical kernels output by the generator. Chapter 5 concludes the thesis by summarizing the results and giving an outlook of possible future work.

Chapter 2

Background and Fundamentals

2.1 The SPIRAL Code Generator

SPIRAL[1] is a program generation system (in other words, a program that generates other programs) used in the domain of digital signal processing (DSP). Its main goal is to automatize the development of high-performance numerical kernels for linear transforms [6, p. 1920], such as the prominent and ubiquitous discrete Fourier transform (DFT).

2.1.1 Key Concepts

Matrix Factorization

In their overview paper [6, p. 1922], the creators of SPIRAL state that DSP algorithms can be represented as matrix-vector multiplications. Both input and output to the algorithm are encoded as vectors and the linear transform to perform is embodied by a particular transformation matrix. Direct application of the matrix-vector product would require $O(n^2)$ operations with regard to a problem size of n. However, clever factorization of the transformation matrix reduces the algorithm's complexity to $O(n \log n)$. Depending on the concrete linear transform, there may exist several different possible factorizations. For this reason, the challenge is to choose the ones leading to optimal algorithmic performance.

SPL and Symbolic Rewriting

SPIRAL uses a concise and expressive mathematical DSL known as Signal Processing Language (SPL) to capture algorithms for DSP transforms. The key idea is as follows [6, pp. 1923 sq.]: matrix factorizations are considered domain-specific knowledge and can be expressed as parameterized *breakdown rules* written in SPL. These rules are fed to the generator by domain experts up front. Using the set of predefined rules, SPIRAL implements a recursive rewriting system. Given a high-level description of the transformation matrix in SPL, the program generator successively applies each fitting breakdown rule to obtain possible factorizations, which may in turn consist of "smaller" (i.e. less complicated)

1 `http://www.spiral.net/`

transformation matrices. Applying the rules recursively leads to a divide-and-conquer approach, eventually delivering a fully expanded (i.e. no more breakdown rules are applicable) description of the factorized transformation matrix. In order to filter out the best of several factorizations, SPIRAL couples this process with empirical autotuning [6, p. 1921].

Example

For example [1, p. 2], a breakdown rule for the famous Cooley-Tukey factorization of a DFT transformation matrix DFT_{mn} of size $m \cdot n$ is given by

$$\mathrm{DFT}_{mn} \rightarrow (\mathrm{DFT}_m \otimes I_n)\ T^n_{mn}\ (I_m \otimes \mathrm{DFT}_n)\ L^{mn}_m.$$

The exact definition of the components mentioned in this rule is not important for our purposes. The key is that the matrix DFT_{mn} left to the arrow $\rightarrow$ is expressed as factorization of other matrices to the right of the arrow, i.a. in terms of the smaller transformation matrices DFT_m and DFT_n, utilizing matrix multiplication and the tensor product "$\otimes$".

2.1.2 Multigrid Methods in SPIRAL

While the original motivation behind SPIRAL was to facilitate the automatic generation of high-performance code for the restricted domain of DSP algorithms, there have been successful efforts to push the generator beyond what was achieved before. As mentioned before, many scientific applications require the solution of PDEs and multigrid-methods are widely used in this context.

Partial Differential Equations

The work [1] of Bolten et al. presents an extension to SPIRAL's present set of breakdown rules to support the generation of a small multigrid solver with a Richardson smoother for a discretized 2-dimensional Poisson equation with Dirichlet boundary conditions. Their paper provides the theoretical foundation of this thesis. Mentioned PDE in its continuous form [1, p. 4] is given by

$$\begin{aligned} -\Delta u(x) &= f(x), \quad x \in \Omega := [0,1]^2, \\ u(x) &= 0, \quad x \in \partial\Omega. \end{aligned}$$

To allow for efficient solving on the machine, the standard procedure is to discretize such equations, yielding a linear system of the form [1, p. 4]

$$Ax = b, \quad A \in \mathbb{R}^{n \times n},\ \ x, b \in \mathbb{R}^n.$$

Multigrid Cycle

Finding a direct solution (e.g. by Gaussian elimination) is costly. But the effort can be reduced substantially by employing an iterative solution method such as Gauss-Seidel or Richardson, starting from an initial guess $x^{(0)}$ and computing a se-

quence of approximations $x^{(k)}$, $k = 0, 1, \ldots$. The error e of such an approximation is given by the solution of the system

$$Ae = b - Ax.$$

After a few iterations, e has not been reduced much but it is smooth. This process is referred to as *pre-smoothing*. The error can now be well represented on a coarser discretization grid (by use of *restriction*), which decreases the number of unknowns considerably. For this reason, obtaining a solution on the coarse grid is now easier. Afterwards, the approximation of the error is transferred back to the fine grid (by use of *prolongation*) and the current approximation of the solution is updated (called *coarse-grid correction*). This process usually introduces undesired high-frequency error components, which can be eliminated by additional subsequent smoothing steps (called *post-smoothing*). This completes one *multigrid cycle*. [1, p. 5]

Multigrid Method

But often times, the coarse problem is still too large to be solved directly [1, p. 5]. Applying the procedure presented above recursively reduces the problem further and further until it is small enough to allow for a direct solution. This strategy is called *multigrid-method*. Implementing multigrid-methods in SPIRAL requires us to reformulate a multigrid cycle in the language of linear algebra, that is, as SPL breakdown rules [1, p. 2]. We limit ourselves to the specification of these rules while omitting the theoretical background.

Mathematical Notation

But first, let us define the mathematical notation [1, pp. 3 sq.] used in SPL. The identity matrix of size $n \times n$ is denoted by the symbol I_n. Similarly, we write 0_n for the $n \times n$ zero matrix. A general matrix M of size $m \times n$ is written as $M_{m \times n}$ but the subscript is dropped if the dimension is clear from the context. Horizontal and vertical stacking of compatible matrices A and B is denoted by

$$[A \mid B] \quad \text{and} \quad \begin{bmatrix} A \\ B \end{bmatrix},$$

respectively. The n-dimensional canonical basis vector with a 1 at the ith location is denoted by e_i^n where $0 \leq i < n$. A scatter matrix $S_{b,s}^{N \times n}$ is an $N \times n$ matrix and a gather matrix $G_{b,s}^{n \times N}$ is an $n \times N$ matrix. They are defined as follows:

$$S_{b,s}^{N \times n} := \left[e_b^N \;\middle|\; e_{b+s}^N \;\middle|\; \ldots \;\middle|\; e_{b+(n-1)s}^N \right] \quad \text{and} \quad G_{b,s}^{n \times N} = \left(S_{b,s}^{N \times n} \right)^\top.$$

The interpretation is that $S_{b,s}^{N \times n}$ scatters the entries x_i of an input vector $x = (x_i)_{0 \leq i < n} \in \mathbb{R}^n$ into a vector $y \in \mathbb{R}^N$ at the locations $b + is$, while setting all

other elements of y to 0. $G_{b,s}^{n\times N}$ performs the inverse operation, that is, it gathers the data from an input vector $x = (x_0, \ldots, x_{N-1})^\top \in \mathbb{R}^N$ starting at base b with stride s, resulting in the vector

$$\left(x_b, x_{b+s}, \ldots, x_{b+(n-1)s}\right)^\top \in \mathbb{R}^n.$$

The tensor product of two matrices $A_{m\times n}$ and $B_{p\times q}$ is defined as follows:

$$A_{m\times n} \otimes B_{p\times q} = [a_{k,l}B], \quad \text{where } A_{m\times n} = [a_{k,l}].$$

The result is a matrix of size $mp \times nq$, where every entry $a_{k,l}$ of A is replaced with the block $a_{k,l}B$. The most notable cases are $A = \mathrm{I}_n$ or $B = \mathrm{I}_n$. To illustrate:

$$I_n \otimes B = \begin{bmatrix} B & & \\ & \ddots & \\ & & B \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} a & b \\ c & d \end{bmatrix} \otimes I_2 = \begin{bmatrix} a & & b & \\ & a & & b \\ c & & d & \\ & c & & d \end{bmatrix}.$$

Tridiagonal matrices are given by

$$\mathrm{Tridiag}_n(a,b,c) = \begin{bmatrix} b & c & & & \\ a & b & c & & \\ & \ddots & \ddots & \ddots & \\ & & a & b & c \\ & & & a & b \end{bmatrix} \in \mathbb{R}^{n\times n}.$$

These “elementary” matrices can be combined to form more complicated expressions using the usual arithmetic operations for matrices, most notably

$$A+B, \quad A\cdot B, \quad A\otimes B, \quad \text{and} \quad \prod_{i=0}^{k-1} A_i$$

for suitable matrices A, B and A_i.

Novel Breakdown Rules

Table 2.1 presents the specification of a rewriting system for the multigrid solver mentioned above. Thereby, a breakdown rule expands a non-terminal symbol into a matrix formula that contains non-terminals itself and termination rules explain a non-terminal in terms of aforementioned elementary matrices.

Entry point of the system is the non-terminal $\mathrm{MGSolvePDE}_{n,\omega,r,m}$. Simply put, it corresponds to m successive applications of a multigrid cycle $\mathrm{MGCycle}_{n,\omega,r}$, which is in turn translated to a Richardson smoother $\mathrm{Richardson}_{n,\omega,r}$ and a subsequent

Table 2.1: SPIRAL SPL breakdown rules for a multigrid solver $\text{MGSolvePDE}_{n,\omega,r,m}$ for an $n \times n$ discretized 2-dimensional Poisson equation with Dirichlet boundary conditions and parameters ω, r and m. The solver uses a Richardson smoother with parameter ω and r iterations and injection as restriction operator. It performs m multigrid cycles. The table has been extracted from [1, p. 7].

$$
\begin{aligned}
\text{MGSolvePDE}_{n,\omega,r,m} &\rightarrow [\mathrm{I}_{n^2} \mid 0_{n^2}] \cdot \left(\prod_{i=0}^{m-1} \text{MGCycle}_{n,\omega,r} \right) \cdot \begin{bmatrix} 0_{n^2} \\ \mathrm{I}_{n^2} \end{bmatrix} \\
\text{MGCycle}_{n,\omega,r} &\rightarrow \text{CGC}_{n,\omega,r} \cdot \text{Richardson}_{n,\omega,r} \\
\text{CGC}_{n,\omega,r} &\rightarrow \begin{bmatrix} \text{CoarseError}_{n,\omega,r} \\ 0_{n^2} \mid \mathrm{I}_{n^2} \end{bmatrix} \\
\text{CoarseError}_{n,\omega,r} &\rightarrow \text{Interpolate}_n \cdot \text{Scatter}_n \cdot \text{Solve}_{n,\omega,r} \cdot \text{Gather}_n \cdot \text{Residual}_n \\
\text{Interpolate}_n &\rightarrow \text{Tridiag}(\sqrt{2}/2, \sqrt{2}, \sqrt{2}/2) \otimes \text{Tridiag}(\sqrt{2}/2, \sqrt{2}, \sqrt{2}/2) \\
\text{Scatter}_n &\rightarrow \mathrm{S}_{1,2}^{n \times (n-1)/2} \otimes \mathrm{S}_{1,2}^{n \times (n-1)/2} \\
\text{Solve}_{n,\omega,r} &\rightarrow \begin{cases} \frac{1}{4}\mathrm{I}_1, & n = 3 \\ [\mathrm{I}_{((n-1)/2)^2} \mid 0_{((n-1)/2)^2}] \cdot \text{MGCycle}_{(n-1)/2,\omega,r} \cdot \begin{bmatrix} 0_{((n-1)/2)^2} \\ \mathrm{I}_{((n-1)/2)^2} \end{bmatrix}, & n > 3 \end{cases} \\
\text{Gather}_n &\rightarrow \mathrm{G}_{1,2}^{(n-1)/2 \times n} \otimes \mathrm{G}_{1,2}^{(n-1)/2 \times n} \\
\text{Residual}_n &\rightarrow [\text{Tridiag}_n(1,-2,1) \otimes \mathrm{I}_n + \mathrm{I}_n \otimes \text{Tridiag}_n(1,-2,1) \mid \mathrm{I}_{n^2}] \\
\text{Richardson}_{n,\omega,r} &\rightarrow \prod_{i=0}^{r-1} \begin{bmatrix} \text{ResidueLaplace}_{n,\omega} & \omega\,\mathrm{I}_{n^2} \\ 0_{n^2} & \mathrm{I}_{n^2} \end{bmatrix} \\
\text{ResidueLaplace}_{n,\omega,r} &\rightarrow \text{Tridiag}_n(\omega, 0.5 - 2\omega, \omega) \otimes \mathrm{I}_n + \mathrm{I}_n \otimes \text{Tridiag}_n(\omega, 0.5 - 2\omega, \omega)
\end{aligned}
$$

coarse-grid correction $\text{CGC}_{n,\omega,r}$. Note the flow of information from *right to left* in the SPL breakdown rules as vectors are applied from right to left. The breakdown for $\text{CoarseError}_{n,\omega,r}$ hard codes injection as choice for restriction. Similarly, the rule Interpolate_n states that linear interpolation is used for prolongation. The non-terminal $\text{Solve}_{n,\omega,r}$ encodes the multigrid recursion. Base cases ($n = 3$) are solved directly, whereas all other cases ($n > 3$) require additional multigrid cycles. The symbol Residual_n is used to compute the residual $b - Ax$ and $\text{Richardson}_{n,\omega,r}$ implements a Richardson smoother with parameter ω and r iterations. Note that Residual_n, Interpolate_n and $\text{ResidueLaplace}_{n,\omega,r}$ set the problem type to be a

Poisson equation and the Dirichlet boundary condition is encoded via the entries of the tridiagonal matrices. [1, p. 7]

This is all mathematical background it takes. The basic implementation of SPL in Scala is presented in Section 3.2. Implementing the rewrite system given by Table 2.1 is discussed in Section 3.3.

2.1.3 SPIRAL in Scala

SpiralS[2,3] is a partial reimplementation of a small subset of the SPIRAL code generator in the programing language Scala[4] which covers the generation of fixed size C code for fast Fourier transforms.

Systematic Construction of Code Generators

In the accompanying publication [3], the authors claim that program generators for high-performance libraries are often merely "ad-hoc collections of standalone programs and scripts that are hard to extend, maintain and reuse". The main goal of their work [3, p. 125] was to (1) analyze already existing generators (i.e. SPIRAL) in order to derive a common and systematic implementation approach, and (2) conduct a case study using the example of SPIRAL to demonstrate how this approach can be realized with high-level programming principles and techniques. Their research on the first point led to the proposal of the implementation approach sketched below [3, p. 126]:

- Describing problem and algorithmic knowledge through one or multiple DSLs of successively lower abstraction levels to allow for various phases of program optimization.
- Specifying certain optimizations and algorithmic choices as rewrite rules on DSL programs.
- Designing parameterized high-level data structures to support the generation of multiple low-level representations and data formats.
- Relying on common infrastructure for recurring low-level transformations, such as unrolling with scalar replacement or selective precomputation.

To achieve these points, any programming environment that provides the required language features can be used. The authors of SpiralS relied on a combination of the programming language Scala and the staging framework LMS, a setup which is also used in this work. The reasons for this particular choice are manifold [3, p. 134]. For instance, Scala's object-oriented programming paradigm turned out to be beneficial for structuring the generator and implementing the DSLs, while

2 `http://www.spiral.net/software/spiral-scala.html`
3 Source code available at `https://github.com/GeorgOfenbeck/SpiralS`
4 It is assumed that the reader already possesses a basic understanding of Scala. Some of its more advanced features are explained as they come up on the way.

functional programming concepts were used to express mathematical algorithms. Furthermore, the support for pattern matching and extractors proved beneficial with regard to DSL rewriting and translation.

The lessons learned and insights gained from the SpiralS project, i.a. which software architectures and programming techniques to use when building the program generator, served as major guidance for the implementation of this work, which will be discussed extensively in Chapter 3.

To support program generation, the staging framework LMS and the closely related Scala-Virtualized compiler are put to work. They are integral part in the implementation and are presented in greater detail in the following sections.

2.2 Lightweight Modular Staging

Lightweight Modular Staging (LMS) is a dynamic multi-stage programming (MSP) approach for the Scala language, implemented as a compiler framework and accessible at the library level. Its main goal is to simplify the process of developing internal DSLs and domain-specific compilers. Extensive information about LMS can be found in Tiark Rompf's PhD thesis [7]; tutorials and example programs are available online [8]. A brief introduction is given below.

2.2.1 Key Principles

Type-Driven Staging

Many dedicated MSP languages such as MetaOCaml rely on syntactic annotations (in particular quasi-quotation) to mark program parts for staging [9, p. 121]. The first idea of LMS is to break with this tradition and to use *type signatures* for distinguishing between stages instead. For this purpose the higher-kinded type `Rep[_]` is provided. The programmer has the responsibility to declare which stage an expression should belong to by explicitly giving it a certain type [10, p. 102], as demonstrated in the following Scala fragment:

```
val x: Int = ...
val y: Rep[Int] = ...
```

Variable `x` is a plain `Int`, hence its value is computed during the first stage, i.e. when the program generator (the *meta program*) is run. Consequently, `x` becomes a constant in the second stage (the *object program*). It is often said that `x` is *static*. In contrast, variable `y` is of type `Rep[Int]`. The type constructor `Rep[_]` turns `y` into a staged expression. The evaluation of `y` is deferred until the second stage takes place, i.e. when the generated code is executed, where `y` appears as a computation with result type `Int`. Hence, `y` is said to be *dynamic*.

An important observation is that the generated code does not pay any runtime cost for static variables [10, p. 102] due to *precomputation*. Moreover, program generator and DSL code are both part of the same program, expressed in terms of the same syntax. This principle is known as *linguistic reuse* and has many benefits [7, p. 23]. To name one, since types are used to distinguish between computational phases, we can leverage the Scala type system to automatically ensure that well-typed meta programs lead to well-typed object programs as well.

Semantic Composition of Staged Code

Now that it is known how to make binding times of expressions explicit, the next question that arises is how to manipulate staged program fragments. This leads to the second idea of LMS: operations on staged types are distributed as (overloaded) operators, which are packaged as components in the form of Scala mix-in traits [9, p. 122]. In conjunction with the first idea, this technique makes it possible to combine staged code in a semantic way, rather than simply expanding it syntactically [10, p. 94].

Example: Staged Power Function

In order to clarify, let us take a look at the classic introductory example for staging [7, pp. 23 sq.], namely the power function $b \mapsto b^n$ where $n \in \mathbb{N}$. A straight-forward implementation in Scala is given below:

```
def power(b: Double, n: Int): Double =
  if (n == 0) 1.0 else b * power(b, n - 1)
```

Imagine the scenario that a program will take many different numbers to the same power. Therefore, we would like to specialize this function for fixed exponents of `n`. In other words: `n` is statically known while the base `b` is only dynamically available. Thus, we want to turn `b` into a staged expression. In LMS, this circumstance is reflected by changing the declared type of `b` from `Double` to `Rep[Double]`. Since a part of the function's input is now dynamic, the output can only be computed dynamically as well. Hence, the new return type must be `Rep[Double]` instead of just `Double`. This gives the implementation listed below:

```
def power(b: Rep[Double], n: Int): Rep[Double] =
  if (n == 0) 1.0 else b * power(b, n - 1)
```

Staging the power function introduced only a comparatively low amount of syntactic overhead: the function body remains unchanged, merely the signature had to be slightly tweaked.

Composing Modules

This code, however, does not compile yet. One problem is that both `b` and `power(b, n - 1)` are no longer of type `Double`, which means they cannot be combined using ordinary multiplication `*`. This is where component technology and semantic composition come into play. LMS provides several traits that define operations on staged types as overloaded operators. For example, basic arithmetic on staged primitives (such as `Rep[Double]`) is bundled in trait `PrimitiveOps`.

This functionality can either be accessed directly through *path-dependent types* and `import` statements:

```
val p: PrimitiveOps = new ... // instantiation of implementation trait
import p._                    // bringing members into scope, i.a. Rep[_]
def power(b: Rep[Double], n: Int): Rep[Double] = ...
```

Alternatively, it can be requested indirectly using *selftype annotations*, which can be seen as "the Scala way" of specifying needed components for dependency injection [11, p. 660]. These annotations are preferred over path-dependent types because they allow for higher modularity. In order to use selftypes, we have to wrap the power function in its own trait, which we call `Power`. This way, `Power` becomes a component itself and may in turn be reused as a mix-in for other traits.

```
trait Power { this: PrimitiveOps =>
  def power(b: Rep[Double], n: Int): Rep[Double] = ... }
```

Special attention should be payed to the selftype annotation

```
this: PrimitiveOps =>
```

in the first line of the definition of `Power`. This very annotation causes that, practically speaking, an instance of `Power` cannot be created without also mixing in a concrete (but unspecified) instance of trait `PrimitiveOps` [9, p. 122].

In any way, we have now gained access to the operations supplied by trait `PrimitiveOps`, which does not only provide a suitable definition for multiplication, but also implicitly converts plain `Doubles` like `1.0` into their staged counterparts of type `Rep[Double]`. Ultimately, the code listed above type checks.

Splitting Interface and Implementation

Another important aspect of LMS is that DSLs are conceptually split into two parts [10, p. 96]: an *interface* and one or possibly more corresponding *implementations*. Both parts can be combined from traits to form a complete definition of a DSL. It is important to realize that DSL programs are written *only* in terms of the given interface, while being completely unaware of the underlying implementation. For example, aforementioned staged `power` function relies on the interface `PrimitiveOps` but is agnostic of its implementation `PrimitiveOpsExp`.

Not only is this conceptual separation demanded by common software engineering principles, e.g. to swap out an implementation for another without affecting client code. But perhaps even more urgent is the concern for *safety* [7, p. 92]. If it were possible for a DSL program to inspect and reason about its own structure (i.e. implementation), optimizing rewrites that maintain semantical but not structural equality could no longer be performed without risking to change the meaning of the program.

2.2.2 Inner Workings

The idea of separating interface and implementation is so fundamental that it can be observed throughout the entire inheritance hierarchy of LMS, which we shall now examine.

Representing Staged Code

At the root of the hierarchy is the interface trait `Base` [7, p. 57]. It declares the well-known type constructor `Rep[T]` and an abstract method named `unit`, which is intended to lift static values of type `T` and produce dynamic ones of type `Rep[T]`.

```
trait Base {
  type Rep[T]
  protected def unit(x: T): Rep[T] }
```

Every Scala program that wants to make use of staging has to mix in `Base`. The interpretation of `Rep[T]` is that it *represents* a computation that yields a result of type `T` in the next program stage [9, p. 122]. Interestingly enough, `Rep[T]` is merely an abstract type and therefore the concrete intermediate representation (IR) of staged expressions is left open.

Strings

In fact, it is possible to treat staged program parts as plain strings by defining

```
type Rep[T] = String
```

However, this is arguably not the most useful representation since we would like to obtain a more analyzable structure [9, p. 124], e.g. to check whether the object program warrants certain guarantees or to perform optimizations before unparsing to actual code.

Expression Trees

For this reason, the LMS core library gives a concrete IR in trait `BaseExp` [7, p. 58], which is the implementation counterpart of `Base`:

```
trait BaseExp extends Base with Expressions {
  type Rep[T] = Exp[T]
  protected def unit(x: T) = Const(x) }

trait Expressions {
  // atomic expressions
  abstract class Exp[T]
  case class Const[T](x: T) extends Exp[T]
  case class Sym[T](id: Int) extends Exp[T]
  // composite definitions: Exp + Exp, Exp * Exp, ...
  abstract class Def[T]
  // statements: val x = Def
  case class Stm[T](sym: Sym[T], rhs: Def[T])
  // block scopes: { Stm; Stm; ...; Stm; Exp }
  case class Block[T](stms: Stm[_], res: Exp[T]) }
```

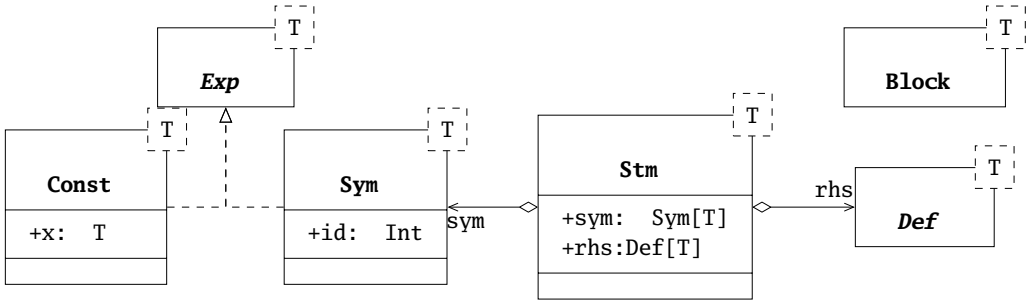

Figure 2.1: Expression trees as defined by `BaseExp`

This listing deserves further explanation. First of all, `BaseExp` extends the interface `Base`. It defines `Rep[T]` in terms of expression trees[5] `Exp[T]`. Therefore, `BaseExp` also mixes in the `Expressions` trait, which contains a concrete implementation of expression trees.

We observe four kinds of expressions: (1) atoms (subclasses of `Exp[T]`), which are either constants `Const[T]` or symbols `Sym[T]`, (2) composite definitions `Def[T]`, (3) statements `Stm[T]`, and (4) block scopes `Block[T]`. Constants convert static values of type `T` into their staged representation of type `Exp[T]`. For this reason, `BaseExp` defines `unit` in terms of `Const`. Symbols, on the other hand, embody unique identifiers referring to definitions. The latter are obtained by combining atoms into more complex expressions utilizing (often times user-defined) operators. No concrete subclasses of `Def[T]` are given because only DSL authors themselves can have the knowledge to provide meaningful data types. The link between definitions and symbols is established by statements `Stm[T]`. Finally, (nested) scopes consisting of a series of statements `stms` and a final return value `res` are modeled by class `Block[T]`. The IR is also depicted in Figure 2.1.

An Extensible IR

Note that `Def[_]` is left abstract by intent. It is encouraged to extend the IR by providing own subclasses of `Def[_]` to further refine the semantics of DSL expressions. Generic optimizers (e.g. the ones provided by the framework, c.f. Page 36) may view the IR in terms of its base nodes (`Exp[_]` and `Def[_]`) while user-defined domain-specific optimizers can leverage the richer semantics to view the program on a higher level of abstraction [10, p. 97].

Unparsing

Code generation is an explicit operation and will be presented in greater detail in Section 3.8.

5 Actually, LMS uses a directed graph ("sea of nodes") that can be accessed via a tree-like interface. [9, p. 124]

2.2.3 Important Utility Functions

Now that we have a basic understanding of LMS' internal structure, let us examine the `Expressions` trait in greater detail, focusing on its most important and practical utilities in particular.

From Syms to Defs

Besides the abstract class `Def`, there is also a corresponding *object* with the same name `Def`, i.e.

```
trait Expressions { ...
  abstract class Def[T]
  object Def { ... } }
```

Objects can be thought of as singletons which are Scala's analogue to the concept of static members of a class in Java. The object `Def` is intended to act as an *extractor* for pattern matching [7, p. 67]. Given a symbol `sym`, it facilitates the navigation to its associated definition as follows:

```
sym match { case Def(d) => ... }
```

After the arrow `=>`, the definition which `sym` refers to is bound to the name `d`. This mechanism comes in handy when traversing and inspecting the IR, i.a. to implement domain-specific rewrites or to translate between different DSLs.

From Defs to Syms

But how are definitions and symbols glued together in the first place? The answer is, there exists a function `toAtom` for this sole purpose.

```
implicit def toAtom[T](d: Def[T]): Exp[T] = ...
```

As its name suggests, `toAtom` takes a composite definition `d` and converts it to an atomic expression (i.e. a symbol referring to `d`). Behind the curtain [7, p. 67], LMS keeps track of already encountered definitions. If `toAtom` finds a previous definition that is structurally equivalent to the given one, a corresponding symbol has already been created, which is then simply returned. Otherwise, a fresh new symbol is produced and linked with the current definition first (by creating a corresponding statement `Stm`), before it is then returned.

Common Subexpression Elimination

Since a definition may only refer to other definitions via their associated symbols, every definition will be named [7, p. 58], effectively allowing for common subexpression elimination (CSE). This is an important optimization technique that prevents unnecessary duplication of code. Note that CSE is implemented completely generic and independent of any particular DSL.

Furthermore, it should be highlighted that `toAtom` is marked with the `implicit` keyword. This qualifies the function for so called *implicit conversions* [11, ch. 21.2]. Simply put, this causes that the conversion from definitions to symbols will take place automatically.

2.3 Scala-Virtualized

Scala is a purely object-oriented language, e.g. operators are not a special kind of syntax but implemented as ordinary methods [11, ch. 5.4]. This is feasible because valid identifiers may not only contain alphanumeric characters but also symbolic ones, or even a combination of both. So `+`, `unary_!` and `::` are in fact legal method names. Furthermore, it is possible to use *any* method (no matter whether it has a symbolic name or not) in operator notation. For example, the `foreach` combinator on lists can be invoked in two ways:

```
List(1, 2, 3).foreach(println) // as method call
List(1, 2, 3) foreach println  // infix operator
```

Both expressions are equal. Vice versa, the computation `1 + 2` is just syntactic sugar for `1.+(2)`. In other words: There is a method named `+` defined in class `Int` that takes an `Int` as argument and returns an `Int` as result:

```
abstract class Int { ...
  def +(x: Int): Int // just to appease the compiler }
```

This high amount of syntactic flexibility proves beneficial when implementing embedded DSLs. Nevertheless, there are certain shortcomings of plain Scala which are addressed in *Scala-Virtualized* [12], a small set of extensions to the regular Scala compiler meant to further enhance its DSL hosting capabilities. In fact, Scala-Virtualized has originally been developed for use in LMS but is now used in several other projects as well. A discussion of its main features follows below.

2.3.1 External Functions

In Section 2.2.2 it was mentioned that all (custom) operations on staged types, such as the multiplication $*$ seen in Section 2.2.1, are realized as operators on type `Rep[_]`. Taking the introductory paragraph of this section into consideration, one may suspect these operators to be implemented as instance methods of `Rep[_]`. But then again, this is not possible because `Rep[_]` is just an abstract type constructor independent of an underlying implementation.

To overcome this problem, Scala-Virtualized adds external functions [7, pp. 33 sq.] to the repertoire, which provide an easy way of defining methods (operators) on objects outside of their class.[6] The idea is similar to operator overloading in C++. Here's how trait `PrimitiveOps` declares multiplication on staged `Doubles`:

6 To some extent, the same effect can be achieved in regular Scala, too, albeit in a more cumbersome way using implicit functions. External functions may even override already existing members of a class, which is not possible with implicit functions.

```
trait PrimitiveOps {
  def infix_*(lhs: Rep[Double], rhs: Rep[Double]): Rep[Double]
  // further functions for addition, subtraction, etc. }
```

This function can be invoked in three ways [7, p. 34]:

```
infix_*(lhs, rhs)
lhs.*(rhs)
lhs * rhs
```

2.3.2 Language Virtualization

Regular Scala allows for extension of built-in language constructs, in particular `for`-comprehensions[7]. Such expressions are translated into calls of `map`, `flatMap`, `withFilter` and `foreach` [11, ch. 23.4]. For example, a `for`-comprehension involving two generators such as

```
for { x <- List(1, 2, 3, 4, 5)
      y <- List('a', 'b', 'c')
} yield (x, y)
```

is tranrsformed by the compiler into

```
List(1, 2, 3, 4, 5) flatMap { x =>
  List('a', 'b', 'c') map { y => (x, y) } }
```

The point is that every user-defined data type that provides suitable combinators can be used with `for`-comprehensions, making for a seamless integration as first-class citizen.

Scala-Virtualized generalizes this idea by redefining all language constructs as virtual methods to make them overloadable by the programmer. This principle is called *language virtualization*[8] [7, p. 5] and is especially helpful in the context of DSLs in order to create an idiomatic syntax that closely resembles the given domain. Overloadable constructs [13] (loops, assignments and others) are defined in trait `EmbeddedControls`, which is mixed into `Predef`[9] and is therefore implicitly available in every Scala program. If no custom implementation of language constructs is given, the usual semantics apply.

For instance, the if-then-else control structure

```
if (c) t else e
```

is translated into the method

7 One may notice a close resemblance to Haskell's do-notation.
8 Hence the name "Scala-Virtualized".
9 It is the analogue to the module `Prelude` in Haskell.

```
__ifThenElse(c, t, e)
```

and behaves exactly as one would expect. Several practical redefinitions are imaginable, e.g. selecting between two branches of control flow based on a dynamic condition. For illustration purposes, a somewhat peculiar redefinition is given below.

```
override def __ifThenElse[T](c: => Boolean, t: => T, e: => T): T =
  c match { case true => e; case false => t }
println(if (true) "success" else "failure")
```

The control structure requires three arguments, passed via call-by-name: a condition `c`, a then-branch `t` and an else-branch `e`. If the condition evaluates to `true` the else-part is executed, otherwise the then-part is picked. For this reason, the program prints `"failure"` and not `"success"`.

2.3.3 Reifying Static Information

Finally, Scala-Virtualized is able to reify static information and make it available in the generated program via `Manifests` and `SourceContexts`. They will be discussed shortly.

Implicit Parameters

Behind the scenes, a language feature called *implicit parameters*[10] [11, ch. 21.5] is put to work. The general idea is as follows: Given a curried function whose *last* parameter list is marked with the `implicit` keyword, the caller only needs to supply the arguments for the preceding parameter lists. The compiler then tries to pass the remaining arguments for the implicit parameter list. To avoid ambiguities and non-deterministic behavior, such implicit arguments are not summoned from the nether land. They must already exist as *unique* value definitions in the current context and be tagged `implicit` themselves. However, there are two exceptions to this rule: when it comes to `Manifests` and `SourceContexts`, the compiler may automatically create suitable instances and insert them on demand [7, p. 37].

Type Information

`Manifests` provide runtime descriptors of data types. They see frequent use in regular Scala programs in those cases where static information about polymorphic types is lost due to type erasure on the Java Virtual Machine (JVM). In the context of internal DSLs they can be useful to generate efficient specialized code, e.g. when the generator uses arrays of generic type, but the resulting DSL program should be specialized to arrays of primitive types [7, p. 37].

An example for a function requiring a `Manifest` of element type `T` is given below:

10 According to Scala's creators, "implicit parameters were motivated by Haskell's type classes; they achieve analogous results in a more classical object-oriented setting." [11, p. 62]

```
def func[T](x: T)(implicit m: Manifest[T]) = ...
```

Notice the implicit parameter list `(implicit m: Manifest[T])` in the declaration of `func`. Within its body, the runtime manifest of `T` can be accessed via the identifier `m`. Callers only need to provide the arguments for the first parameter list of `func`, which solely consists of `x` in this case, e.g. `func(42)`.

Often times, *context bounds* [11, ch. 21.6] are used as syntactic sugar for implicit parameter declarations. The following definition of `func` is equivalent to the one listed above:

```
def func[T: Manifest](x: T) = { val m = implicitly[Manifest[T]]; ... }
```

The notation `[T: Manifest]` is the context bound. It introduces a type parameter `T` and an unnamed implicit argument of type `Manifest[T]`, which has to be retrieved by the `implicitly` function.

Source Information

Implicit `SourceContext` objects provide a generated program with static source code information which is otherwise impossible to recover once the object program is run [7, pp. 37 sq.]. Programmers can request `SourceContexts` for methods in the same way as `Manifests` using an additional implicit parameter:

```
def func[T](x: T)(implicit pos: SourceContext) = ...
```

Here, the source context of `func`'s invocation site, such as the file name, line number and character offset, is available as `pos`. A common use case is to chain implicit `SourceContexts` to reflect the static call path, thus allowing for better error messages and improved debugging of DSL programs.

Chapter 3

Implementation

This chapter covers the implementation of the code generator. We first provide a small overview of the generator architecture. Afterwards, we proceed with the implementation of the first DSL, namely SPL. Next, we further refine SPL by implementing domain-specific optimizations. Subsequently, we tackle the implementation of the second DSL, called CIR. This is followed by a detailed description of the translation process between SPL and CIR. Then, we demonstrate how to boost efficiency of the generated code by means of stencil computations. Finally, we show how to unparse to actual C code.

3.1 Overview of the Generator Architecture

This introductory section gives an overview of the generator architecture. Inspired by the original SpiralS prototype, the fundamental design principle is to use several DSLs operating on different levels of abstraction. To be precise, we employ (1) SPL to capture algorithmic and domain-specific knowledge, and (2) CIR as an internal representation of C. Notice that each DSL operates on a different level of abstraction and is thus fitted for different kinds of structural, algorithmic and performance-based optimizations. Also, both DSLs are embedded into Scala and will therefore reuse large portions of its syntax and grammar.

As illustrated by Figure 3.1 on Page 20, the input to the generator is a fully fixed static specification $\text{MGSolvePDE}_{n,\omega,r,m}$ of the PDE to solve. A suitable algorithm in SPL is obtained by recursively applying the breakdown rules of Table 2.1 to the input. High-level structural optimizations are performed on-the-fly. Afterwards, the structurally optimized SPL program is translated into CIR. This lowers the algorithm's level of abstraction. Low-level optimizations, such as algebraic simplification or common subexpression elimination, take place while constructing the CIR program. Last but not least, the IR graph representing the optimized CIR algorithm is unparsed to C code.

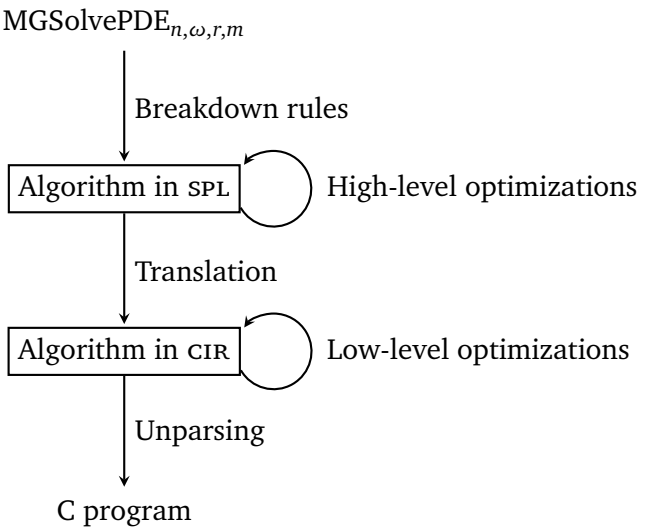

Figure 3.1: Program generator architecture for our small multigrid solver

3.2 Signal Processing Language

The implementation of SPL is split into two parts: a representation of the matrices found in Table 2.1, and the corresponding operators to manipulate them. We start with matrices.

3.2.1 Matrices

Superclass SPL

As mentioned earlier, Scala adheres to the object-oriented programming paradigm. Hence, every SPL matrix is naturally represented as a regular Scala class that derives from a common abstract superclass SPL:

```
sealed abstract class SPL(val rows: Int, val cols: Int) {
  require(rows > 0 && cols > 0)
  def this(n: Int) = this(n, n) }
```

It captures two significant properties of a matrix, namely its number of rows `rows` and columns `cols`. Furthermore, it also enforces that these are only set with sensible values (i.e. positive integers to represent a dimension). For this purpose, the `require` function is used, which is part of the Scala standard library. It throws an `IllegalArgumentException` if the given condition is not satisfied. As a result, an instance of SPL will not be constructed in case of a failing requirement. We also define a secondary constructor to make the creation of quadratic matrices more convenient. As we are going to perform (structural) pattern matching on matrices, we declare SPL to be `sealed`. This causes that specializations of SPL can

only be added in the same source file containing SPL itself and therefore enables the compiler to issue a warning in case of non-exhaustive patterns [11, p. 323].

Concrete Subclasses

In similar fashion, every subclass of SPL introduces its own additional properties and invariants. For example, the matrix $\text{MGSolvePDE}_{n,\omega,r,m}$ is modeled by class `MGSolvePDE(n, omg, r, m)`. It makes sure that the number of iterations r for the Richardson smoother and the number of multigrid cycles m are both positive. Moreover, the size n of the square grid must be exactly one less than a power of two, or more formally, be part of the set

$$N \coloneqq \{2^k - 1 \mid k \geq 2\} = \{3, 7, 15, 31, \ldots\}.$$

The latter condition can be tested efficiently with a bit of bit-twiddling. In binary, every $n = 2^k - 1 \in N$ is composed solely of k occurrences of the digit 1. Its successor $n+1$ starts with a leading 1, followed by nothing else but k times the digit 0. Hence, taking the bitwise logical conjunction of n and $n+1$ yields the number 0.

$$\begin{array}{rcll} & & & \overbrace{}^{k \text{ times}} \\ n & = & 2^k - 1 & \simeq \quad 111\ldots111 \\ n+1 & = & 2^k & \simeq \quad 1\,000\ldots000 \\ \hline n \,\&\, (n+1) & = & & 0\,000\ldots000 \end{array}$$

If the result is not 0 we are dealing with an undesired number that is not element of N. In Scala, this constraint is expressed as

```
require((n > 2) && (n & (n + 1) == 0))
```

Putting it all together, we obtain the following code:

```
case class MGSolvePDE(private val n: Int, omg: Double, r: Int, m: Int)
    extends SPL(n * n) {
  require((n > 2) && (n & (n + 1) == 0) && (r > 0) && (m > 0)) }
```

All other SPL matrices are represented analogously. They are listed in Table 3.1 together with the corresponding SPL subclass. The next step is to define operations on SPL to allow for actual useful work with the DSL.

3.2.2 Operators

The usual object-oriented approach suggests that every operator should be encoded as instance method of SPL. This would allow us to compute complex expressions, such as breakdowns, statically during generator time. However, this implementation falls short of the fact that the input vector is only *dynamically* available, i.e. during the execution of a program that is yet to be generated. Hence,

Table 3.1: Mapping of SPL matrices to their appropriate SPL counterparts. The expressions above the thin line are non-terminal, whereas the matrices below that line are terminal.

SPL matrix	SPL subclass	rows × columns
$\text{MGSolvePDE}_{n,\omega,r,m}$	`MGSolvePDE(n: Int, omg: Double, r: Int, m: Int)`	$n^2 \times n^2$
$\text{MGCycle}_{n,\omega,r}$	`MGCycle(n: Int, omg: Double, r: Int)`	$2n^2 \times 2n^2$
$\text{CGC}_{n,\omega,r}$	`CGC(n: Int, omg: Double, r: Int)`	$2n^2 \times 2n^2$
$\text{CoarseError}_{n,\omega,r}$	`CoarseError(n: Int, omg: Double, r: Int)`	$n^2 \times 2n^2$
Interpolate_n	`Interpolate(n: Int)`	$n^2 \times n^2$
Scatter_n	`Scatter(n: Int)`	$n^2 \times (n-1)^2/4$
$\text{Solve}_{n,\omega,r}$	`Solve(n: Int, omg: Double, r: Int)`	$(n-1)^2/4 \times (n-1)^2/4$
Gather_n	`Gather(n: Int)`	$(n-1)^2/4 \times n^2$
Residual_n	`Residual(n: Int)`	$n^2 \times 2n^2$
$\text{Richardson}_{n,\omega,r}$	`Richardson(n: Int, omg: Double, r: Int)`	$2n^2 \times 2n^2$
$\text{ResidueLaplace}_{n,\omega,r}$	`ResidueLaplace(n: Int, omg: Double)`	$n^2 \times n^2$
$\text{S}_{b,s}^{r \times c}$	`S(rows: Int, cols: Int, base: Int, stride: Int)`	$r \times c$
$\text{G}_{b,s}^{r \times c}$	`G(rows: Int, cols: Int, base: Int, stride: Int)`	$r \times c$
I_n	`I(n: Int)`	$n \times n$
0_n	`Zero(n: Int)`	$n \times n$
$\text{Tridiag}_n(a,b,c)$	`Tridiag(n: Int, a: Double, b: Double, c: Double)`	$n \times n$

the resulting vector can only be computed dynamically as well. Consequently, one must adapt the implementation for two separate stages of execution.

Staged Operators

We employ LMS for staging. For this reason, the correct approach is to define operators in such a way that expressions of type `Rep[SPL]` are processed. As explained in Sections 2.2 and 2.3, the common workflow is to use external functions instead of ordinary methods, and to make a sharp distinction between interface and implementation.

SPL Interface

In accord with these principles, matrix operations are declared in a separate interface trait `SPLOps`.

```
trait SPLOps extends Base { ... }
```

It mixes in the obligatory `Base` trait which literally serves as a base for every custom DSL. Subsequently, for every operator that is to be used in an SPL program,

a corresponding infix function is provided. For example, matrix addition + is realized as follows:

```
def infix_+(lhs: Rep[SPL], rhs: Rep[SPL])
    (implicit pos: SourceContext): Rep[SPL]
```

Besides the two staged matrix expressions `lhs` and `rhs`, the operator also requires an implicit `SourceContext` in a separate parameter list, making `infix_+` a curried function. The latter argument, however, is passed automatically by the compiler so that clients need not be concerned with providing it themselves. Hence, using the partially applied function `lhs + rhs` in client code suffices. Implicit parameters entail quite a bit of syntactic overhead and they only portray a minor implementation detail. For this reason, they are tacitly omitted from now on.

Internal Aliases

In order to achieve even looser coupling between interface and implementation, we introduce an internal alias for every matrix operation, for instance:

```
def infix_+(lhs: Rep[SPL], rhs: Rep[SPL]): Rep[SPL] = spl_plus(lhs, rhs)
protected def spl_plus(lhs: Rep[SPL], rhs: Rep[SPL]): Rep[SPL]
```

The name `infix_+` is intended for use in DSL programs, whereas `spl_plus` is used in the program generator *only*, hence the `protected` modifier to hide it from clients. This design allows us to change the syntax of operators in only one place without affecting the rest of the generator's code base.

Matrix addition exemplifies the general implementation strategy which also applies to all other operators. Table 3.2 summarizes which operators are available in SPL programs.

Operator Precedence

There is one important note to make. When using external functions such as `infix_+` or `infix_*` in infix notation, Scala's usual rules for operator precedence [11, ch. 5.9] apply. For example, the symbol * has a higher priority than +. This means that for an SPL expression, in which `A`, `B` and `C` are suitable matrices, the following holds:

```
A + B * C == A + (B * C) == infix_+(A, infix_*(B, C))
```

Syntactic Sugar

What we have achieved so far is that we can easily construct new SPL matrices, and we can use the operators in a natural and idiomatic way. Though, when given two matrices `lhs` and `rhs` of plain type `SPL`, we still have to perform the lifting to `Rep[SPL]` ourselves, using the `unit` function. For example, consider the sum of `lhs` and `rhs`:

```
unit(lhs) + unit(rhs)
```

This poses a slight inconvenience to clients, hence one may add additional overloaded variants for every operator and perform the conversion explicitly within their body:

Table 3.2: Supported matrix operations in SPL programs. Placeholders for matrices are denoted by uppercase letters, whereas lowercase letters stand for scalar values.

Operation	Symbol	Public SPL name	Internal SPL name
Addition of matrices	$L + R$	`infix_+(L, R)`	`spl_plus(L, R)`
Multiplication of matrices	$L \cdot R$	`infix_*(L, R)`	`spl_times(L, R)`
Multiplication of a scalar with a matrix	$s \cdot M$	`infix_*>(s, M)`	`spl_stimes(s, M)`
Tensor product of matrices	$L \otimes R$	`infix_tensor(L, R)`	`spl_tensor(L, R)`
Horizontal stacking of matrices	$[L \mid R]$	`infix_hs(L, R)`	`spl_hstack(L, R)`
Vertical stacking of matrices	$\begin{bmatrix} T \\ B \end{bmatrix}$	`infix_vs(T, B)`	`spl_vstack(T, B)`
Exponentiation of a matrix	$\prod_{i=0}^{n-1} M$	`infix_pow(n, M)`	`spl_power(n, M)`

```
def infix_+(lhs: SPL, rhs: Rep[SPL]) = spl_plus(unit(lhs), rhs)
def infix_+(lhs: Rep[SPL], rhs: SPL) = spl_plus(lhs, unit(rhs))
def infix_+(lhs: SPL, rhs: SPL)      = spl_plus(unit(lhs), unit(rhs))
```

Not only is this approach tedious but it also obscures the implementation of the generator with uninteresting and redundant details. Hence, an easier and more elegant solution is to provide an implicit conversion function instead:

```
implicit def liftSPL(i: SPL): Rep[SPL] = unit(i)
```

This is all it takes. Now, it is no longer necessary to call `unit` explicitly, the compiler will insert the appropriate call implicitly for us.

SPL Implementation

Moving on with the implementation, trait `SPLOpsExp` gives a concrete definition of `SPLOps` as follows:

```
trait SPLOpsExp extends SPLOps with BaseExp { ... }
```

Notice that `BaseExp` is also mixed in so that the members inherited from `Base` are well-defined. Concerning the implementation of matrix operators, the idea is to create a new composite node in the IR whenever the corresponding function is invoked. We provide specific nodes for every kind of operator. When analyzing the IR, this helps us to accurately trace back which operations have been performed and will provide us with viable information when applying domain-specific optimizations. As an example, matrix addition `spl_plus` is represented by class `Plus`.

```
trait SPLOpsExp extends SPLOps with BaseExp {
```

```
  case class Plus(lhs: Exp[SPL], rhs: Exp[SPL]) extends Def[SPL]
  /* additional inner classes for the remaining operators */ }
```

`Plus` stores the left and right argument of the addition, which are now of type `Exp[SPL]` instead of `Rep[SPL]`. For the compiler, this change is insignificant because `Exp[T]` simply aliases `Rep[T]`. But for humans, it serves as a gentle reminder that we are now working on the implementation part of SPL. Since `Plus` models a composite operation, it extends `Def[SPL]`.

Avoiding illegal expressions

However, a crucial point is to check whether it is legal to perform a composite operation in the first place. The operands must be compatible with each other, that is to say, their dimensions must match. This decision cannot be made in general, but one has to take the currently considered operator into account. For example, matrix addition requires its left and right argument to have the same row and column dimension, while matrix multiplication demands the left expression to have same number of columns as the number of rows of the right expression.

Refining the IR

This concern highlights the need for an easy way of accumulating and extracting domain-specific knowledge in the IR. Thus, the specialization `SPLDef` of `Def[SPL]` is created:

```
trait SPLDef extends Def[SPL] {
  def rows: Int
  def cols: Int }
```

It stores the row and column dimension of a composite expression directly in the corresponding IR node, making it readily available at our fingertips. Otherwise, we would have to retrieve this information from atoms and recompute the dimensions over and over again. Speaking of retrieval, we also write an auxiliary function `splDim` which tells the dimensions of a given SPL expression, returning them as a pair of rows and columns.

```
def splDim(arg: Exp[SPL]): (Int, Int) = arg match {
  case Const(spl: SPL) => (spl.rows, spl.cols)
  case sym: Sym[_] => sym match {
    case Def(s: SPLDef) => (s.rows, s.cols)
    case _ => throw new IllegalArgumentException }
  case _ => throw new IllegalArgumentException }
```

`splDim` demands an IR node of type `Exp[SPL]`, which is either a `Const` or a `Sym`. In the former case, `Const` is just a wrapper around an `SPL` instance `spl` whose fields `rows` and `cols` can be accessed directly. In the latter case, the pattern extractor `Def` is used to find the concrete instance `s` of `SPLDef` with which the symbol `sym` is associated. Afterwards, the dimensions can be extracted from `s`. In all other cases,

the given node does not represent an SPL expression and we have no choice but to throw an `IllegalArgumentException`.

After that, the implementation of all composite nodes, like `Plus`, needs to be adapted to the recent changes:

```
class Plus(val lhs: Exp[SPL], val rhs: Exp[SPL],
  override val rows: Int, override val cols: Int) extends SPLDef
```

We now extend the more refined `SPLDef` instead of `Def[SPL]`. For this reason, the constructor of `Plus` takes two more arguments (`rows` and `cols`). Both of them are actually defined as *fields* (hence the `val` keywords) that override the corresponding parameterless methods declared in `SPLDef` (hence the `override` modifiers). This is common practice in Scala and naturally arises from the so called *uniform access principle* [11, p. 817]. It states that both fields and methods should be accessible via a uniform syntax so that clients are unable to tell how services are implemented (as computation or via storage).

Ensuring Correctness

Returning to the original goal of avoiding illegal expressions, what still remains to be done is to validate the dimensions before actually creating a composite node. In that respect, the primary constructor of `Plus` is declared `private`:

```
class Plus private (val lhs: Exp[SPL], val rhs: Exp[SPL],
  override val rows: Int, override val cols: Int) extends SPLDef
```

This causes that the constructor can only be accessed from within the class itself or by its *companion object* [11, ch. 4.3], which is created in the next step. Remember that a class and its companion object share the same name, e.g. `Plus` in this case. Inside the object `Plus`, a factory method with the name `apply` is placed. The idea is to task `apply` with the responsibility of checking the dimensions before passing the arguments to the constructor of class `Plus`:

```
object Plus {
  def apply(lhs: Exp[SPL], rhs: Exp[SPL]): Plus = {
    val ((rowsL, colsL), (rowsR, colsR)) = (splDim(lhs), splDim(rhs))
    require(rowsL == rowsR && colsL == colsR)
    new Plus(lhs, rhs, rowsL, colsL) } }
```

The factory method extracts the dimensions from the given left-hand `lhs` and right-hand argument `rhs` using the prior defined function `splDim`. Afterwards, it checks whether matrix addition of `lhs` and `rhs` is possible by comparing their dimensions. If and only if the matrices turn out to be compatible with each other, `apply` passes `lhs` and `rhs`, as well as the correct row and column dimension to the constructor of class `Plus`.

Table 3.3: Summary of matrix operators and their appropriate IR nodes. By convention, uppercase letters denote matrices and lowercase ones stand for scalars. The values `rowsL` and `colsL` refer to the number of rows and columns of a matrix `L`. The same holds for `rowsR` and `colsR` for a matrix `R`, as well as `rows` and `cols` for the matrix `M`.

Operation	IR node class	Constraints
`spl_plus(L, R)`	`Plus(L, R, rowsL, colsL)`	`(rowsL, colsL) == (rowsR, colsR)`
`spl_times(L, R)`	`Times(L, R, rowsL, colsR)`	`colsL == rowsR`
`spl_stimes(s, M)`	`ScalarTimes(s, M, rows, cols)`	—
`spl_tensor(L, R)`	`Tensor(L, R, rowsL*rowsR, colsL*colsR)`	—
`spl_hstack(L, R)`	`HStack(L, R, rowsL, colsL + colsR)`	`rowsL == rowsR`
`spl_vstack(T, B)`	`VStack(T, B, rowsL + rowsR, colsL)`	`colsL == colsR`
`spl_power(n, M)`	`Power(n, M, rows, cols)`	`rows == cols && n > 0`

For the remaining IR nodes other than `Plus`, the modus operandi is analogous. Table 3.3 summarizes SPL operators, their corresponding IR node classes and the constraints that need to be satisfied.

More on Syntactic Sugar

Scala treats methods named "`apply`" specially [11, p. 423]. When constructing new instances of class `Plus`, it is possible to use the notation `Plus(lhs, rhs)` as a shorthand for `Plus.apply(lhs, rhs)`. It is worth emphasizing that clients may only invoke the provided factory method. Calling the constructor explicitly via the `new` keyword, i.e. `new Plus(lhs, rhs, rows, cols)`, is forbidden as it is declared `private`.

Structural Deconstruction

While `apply` is used to create new instances of a class, its dual `unapply` serves the purpose of taking them apart again. It suggests itself to turn the companion object of IR node classes, such as `Plus`, into an extractor for pattern matching [11, ch. 26.2] by adding an appropriate definition of `unapply`:

```
object Plus {
  def unapply(arg: SPLDef): Option[(Exp[SPL], Exp[SPL], Int, Int)] =
    arg match {
      case p: Plus => Some((p.lhs, p.rhs, p.rows, p.cols))
      case _ => None }
  def apply(lhs: Exp[SPL], rhs: Exp[SPL]): Plus = ... /* as before */ }
```

The function takes an IR node `arg` of type `SPLDef`. If `arg` conforms to the type `Plus`, its sequence of constructor arguments is returned, wrapped in `Some`. Otherwise, `None` is returned. Therefore, it is possible to inspect the structure of an IR node by the help of pattern matching, for example:

```
arg match {
  case Plus(lhs, rhs, rows, cols) => // calls Plus.unapply(arg)
  /* possibly more patterns */ }
```

Notice the relationship between the constructor of `Plus` and `unapply`:

```
arg == Plus.unapply(arg) match {
  case Some(lhs, rhs, row, col) => new Plus(lhs, rhs, row, col) }
```

and

```
Plus.unapply(new Plus(lhs, rhs, row, col)) == Some(lhs, rhs, row, col)
```

Putting It All Together

Last but not least, we have to wire operator functions and their respective IR nodes together, as demonstrated by matrix addition:

```
protected def spl_plus(lhs: Exp[SPL], rhs: Exp[SPL]): Exp[SPL] =
  Plus(lhs, rhs)
```

Simply put, every matrix operation yields an instance of the corresponding composite node. However, there is more to it than meets the eye. The IR may only consist of atomic nodes since `type Rep[T] = Exp[T]`. Therefore, `spl_plus` ought to return an `Exp[SPL]` but the given node `Plus` is a composite expression of type `Def[SPL]`. Therefore, it is imperative to convert definitions to symbols within the body of operator functions such as `spl_plus`. Recalling Section 2.2.3, we know that the implicit function `toAtom` will kick in to automatically accomplish the desired conversion for us. Under the hood, the Scala compiler changes the code given above into the following, inserting an appropriate call to `toAtom`:

```
protected def spl_plus(lhs: Exp[SPL], rhs: Exp[SPL]): Exp[SPL] =
  toAtom(Plus(lhs, rhs))
```

This fact is important, because not only does this fix the type error, but it also establishes the correspondence between the evaluation order of the program generator and the generated program: “at the point where the generator calls `toAtom`, the composite definition is turned into an atomic value [...], i.e., its evaluation is recorded now and played back later in the same relative order with respect to others” in the DSL program. [7, p. 59]

This concludes the implementation of SPL. We can now turn our attention to the implementation of breakdown rules.

3.3 Breakdown Rules

Thanks to our efforts before, the implementation of SPL breakdown rules is simple. First, a new trait `Breakdowns` is created to house the rules.

```
trait Breakdowns { this: SPLOps => ... }
```

In the spirit of modular mix-in composition, we use the self-type `this: SPLOps =>` to denote that we require the methods defined in the DSL interface `SPLOps`.

A Top-Down Parser

Next, a function `breakdown` is defined inside `Breakdowns`. It implements the rewriting system specified in Table 2.1. The general working principle is as follows [6, p. 1921]: given an arbitrary initial `SPL` matrix `s`, `breakdown` tries to decompose `s` into a semantically equivalent, but structurally different expression consisting of smaller (i.e. less complex) matrices, using one of the parameterized breakdown rules. The small matrices in the resulting expression may in turn be further expanded by applying one of the rules again. This strategy leads to a recursive divide-and-conquer approach that fully expands the expression `s` until it only consists of terminals. Note that, in our case, there is at most one matching rewrite rule to accomplish one recursive step in the breakdown. But usually, there are higher degrees of freedom involved. This means that several different rules may match the given input or there may even be different parameterizations to the very same rule. In this situation, one needs a mechanism to decide which rule to use next, i.a. autotuning or machine learning.

For the remainder of this section, we exemplify the implementation of the breakdown rule $\text{Solve}_{n,\omega,r}$. Out of all the rules in Table 2.1, it is the most interesting one to consider due to its complexity: it distinguishes between a base case and a recursive case.

Signature

Before we start, let us think about the function signature of `breakdown`. A breakdown rule is made of two parts: a non-terminal matrix to the left of the arrow $\rightarrow$ and a corresponding expansion to the right of the arrow. Expressions on the left such as $\text{Solve}_{n,\omega,r}$ or $\text{MGSolvePDE}_{n,\omega,r,m}$ are static in nature. Hence, the input of `breakdown` should be of plain type `SPL`. Being fed with this input, `breakdown` is supposed to return a fully expanded expression. The result represents code that is to be executed during a later program stage. Therefore, it makes sense to choose `Rep[SPL]` as result type of `breakdown`. This leads to the following function declaration:

```
def breakdown(s: SPL): Rep[SPL]
```

Implementation

Now that we are provided with the scaffolding, it is time to flesh out the body of `breakdown`. The key is to inspect the given argument `s` in a pattern match. The matrix $\text{Solve}_{n,\omega,r}$ is represented by class `Solve(n, omg, r)`. As there are two cases in this breakdown rule, we need two patterns in the pattern match as well.

Base Case

Let us first tackle the implementation of the base case, which has a direct resolution:

$$\text{Solve}_{n,\omega,r} \;\rightarrow\; \frac{1}{4}\text{I}_1, \quad n = 3.$$

The corresponding Scala code looks similar:

```
case Solve(3, _, _) => 0.25 *> I(1)
```

The pattern `Solve(3, _, _)` captures the condition of applicability. Not only do we need an instance of class `Solve` but its dimension `n` must also equal the value `3`. Since the two fields `omg` and `r` are not referenced on the right-hand side of the rule, it is not worth assigning special names to them. So, the wildcard character "`_`" is used instead. Finally, the expression `0.25 *> I(1)`, corresponding to $\frac{1}{4}\text{I}_1$, is returned. There are two important points to notice. First, the Scala code on the right-hand side of a rule is fully-fledged SPL code. We may use arbitrary SPL matrices, such as the identity `I(1)`, and operators, such as the scalar multiplication `*>`. Second, there is no need to explicitly lift static values, as the implicit conversion `unit` will automatically take care of this. Hence, writing `0.25 *> unit(I(1))` is indeed formally correct but also unnecessarily noisy.

Recursive Case

The recursive case

$$\text{Solve}_{n,\omega,r} \;\rightarrow\; [\text{I}_{((n-1)/2)^2} \mid 0_{((n-1)/2)^2}] \cdot \text{MGCycle}_{(n-1)/2,\omega,r} \cdot \begin{bmatrix} 0_{((n-1)/2)^2} \\ \text{I}_{((n-1)/2)^2} \end{bmatrix}, \quad n > 3$$

is essentially implemented in the same fashion:

```
case Solve(n, omg, r) if n > 3 =>
  val dim = ((n - 1) / 2) * ((n - 1) / 2)
  val left = I(dim) hs Zero(dim)
  val right = Zero(dim) vs I(dim)
  val mcgycle = breakdown(MGCycle((n - 1) / 2, omg, r))
  left * mcgycle * right
```

There are subtle, but significant differences. For instance, the guard `if n` ↩ `> 3` is used in the pattern to express that only dimensions greater than `3` are allowed. Moreover, `breakdown` calls itself recursively in the second last line to further expand the expression `MGCycle((n-1)/2, omg, r)` before returning the final result.

Abstraction Without Regret

It is important to emphasize that arbitrary Scala features may be used in SPL code without affecting the generated code. The abstraction and runtime overhead of such language features will have been translated away after the first program stage. For example, instead of calculating the static dimension $((n-1)/2)^2$ inline multiple times, we assigned the result to a meaningful local variable `dim`, which

we could then reuse. In the generated code, `dim` will be replaced with the actual result of its evaluation.

The complete definition of `breakdown` is given below:

```
trait Breakdowns { this: SPLOps =>
  def breakdown(s: SPL): Rep[SPL] = s match {
    case Solve(3, _, _) => 0.25 *> I(1)
    case Solve(n, omg, r) if n > 3 => ... // as before
    // further cases are placed here, but omitted for simplicity
    case _ => s /* No rule applicable */ } }
```

For every rule in Table 2.1, there is one corresponding case distinction (only the two cases of $\text{Solve}_{n,\omega,r}$ are listed). Due to the inner workings of pattern matching, specialized patterns need to be placed before more general ones. Finally, to handle expressions for which no breakdown exists, the wildcard pattern `case _ => s` is added as last rule. It matches any given expression and simply returns the lifted variant of the input. This concludes the implementation of breakdown rules.

3.4 Applying Domain-Specific Optimizations

Performance Issues

As seen in Section 3.2, matrix operators generally insert new nodes in the IR whenever they are invoked. Yet, this behavior seems overly eager. For instance, consider multiplications with the identity or zero matrix, such as $A_{n\times n} \cdot I_n$ or $0_n \cdot B_{n\times n}$. Even though A and B are merely unknowns at this point, we do already know the result of the multiplication, even before the DSL program is run: it is $A_{n\times n}$ or 0_n, respectively. This stems from fundamental algebraic rules, such as I_n being the neutral element with regard to matrix multiplication.

Remedy: Eliminating Operations

The general mindset is that the more operations a program has to perform, the longer it takes to complete its execution [3, p. 133]. Since we strive for high performance, it is advantageous to reduce the amount of unnecessary operations as much as possible. With regard to the small example above, this means that instead of registering a superfluous composite operation with LMS, which causes the creation of several IR nodes, we could simply insert a single atomic node. This is safe because the result of the calculation is already fully determined during generator time and therefore the semantics of the generated program are not altered in any way. Notice that we cannot rely on the Scala compiler or LMS to apply these performance tricks for us because neither of them have the domain-specific knowledge to do so. Only we do.

Our task now is to identify the cases where such optimizations are possible, as outlined by the example given above.

Performance-Optimized SPL

Proceeding with the implementation, the proper place to put these optimizations is a separate trait `SPLOpsExpOpt` that specializes `SPLOpsExp`:

```
trait SPLOpsExpOpt extends SPLOpsExp { ... }
```

This way, it is still possible to choose whether it should be made use of the optimizations, just by mixing in one trait or the other. For example, it is imaginable to tailor certain optimizations for specific target platforms. In such cases, it is best to provide a separate trait which is mixed in only when necessary. Figure 3.2 illustrates the current architecture of SPL.

Smart Constructors

Next, a specialization of each matrix operation is created, thus overriding the appropriate function in the super trait. The idea is that every such specialization implements an optimized version of the corresponding operator. In order to do so, it is mandatory to inspect the given arguments and react accordingly. In other words, we create a *smart constructor* for the IR nodes. For instance, to avoid multiplications with the identity or zero matrix:

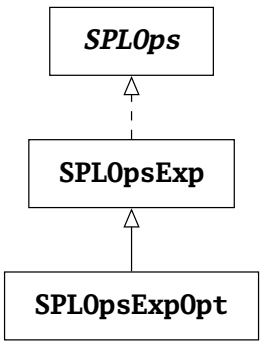

Figure 3.2: SPL architecture

```
override protected def spl_times(lhs: Exp[SPL], rhs: Exp[SPL]) =
  (lhs, rhs) match {
    // multiplications with the zero matrix
    case (Const(Zero(_)), _) => lhs
    case (_, Const(Zero(_))) => rhs
    // multiplications with the identity matrix
    case (Const(I(_)), _) => rhs
    case (_, Const(I(_))) => lhs
    // all other cases
    case _ => super.spl_times(lhs, rhs) }
```

If `lhs` or `rhs` happens to be the zero matrix, this particular argument is yielded. If any of them corresponds to the identity matrix, the other argument is returned. This way, the construction of superfluous nodes in the IR is prevented right from the beginning. Note that the pattern extractor `Const` needs to be employed in order to access the wrapped SPL matrix. In all other cases, i.e. if neither of the arguments match the zero or identity matrix, we delegate to the implementation of `spl_times` in the super trait, which unconditionally produces a composite node:

```
case _ => super.spl_times(lhs, rhs)
```

Although tempting, it is important not to write `lhs * rhs` here, since this notation is merely syntactic sugar for `spl_times` in the *current* trait `SPLOpsExpOpt` and not in the super trait. Hence, we would end up with an infinite recursion.

Maintaining Correctness

This approach is simple, yet effective. However, we still face the problem of avoiding illegal expressions. At the moment, the result of performing the impossible multiplication `I(5) * Zero(1)` is `Zero(1)` because it matches the second pattern of `spl_times`. For the sake of correctness, there is no choice but to validate the dimensions a second time, specifically for the optimizing rewrites.

In Section 3.2, the function `splDim` was used to find the dimensions of a given SPL expression. Yet, this function seems out of place in a pattern match. It is more idiomatic to access the dimensions via a pattern extractor. For this purpose, the object `SPLExp` is introduced in trait `SPLOpsExp`:

```
object SPLExp {
  def unapply(arg: Exp[SPL]): Option[(Int, Int)] = arg match {
    case Const(s: SPL) => Some((s.rows, s.cols))
    case s: Sym[_] => s match {
      case Def(x: SPLDef) => Some((x.rows, x.cols))
      case _ => None }
    case _ => None } }
```

The definition of `unapply` is almost identical to `splDim`. The only difference is that instead of returning a pair directly, the result is now an optional value, represented by Scala's `Option` monad. Thus, a pair of dimensions is wrapped in `Some`, and `None` is returned as opposed to throwing an exception. We simplify implementation of `splDim`:

```
protected def splDim(arg: Exp[SPL]): (Int, Int) = arg match {
  case SPLExp(rows, cols) => (rows, cols)
  case _ => throw new IllegalArgumentException }
```

We may now correct the implementation of `spl_times` by means of `SPLExp` and guards, as exemplified here:

```
override protected def spl_times(lhs: Exp[SPL], rhs: Exp[SPL]) =
  (lhs, rhs) match {
    case (Const(Zero(n)), SPLExp(rows, cols))
      if rows == n && cols == n => lhs // 0_n · B_{n×n} = 0_n
    case (SPLExp(_, cols), Const(I(n))
      if cols == n => lhs              // A_{n×n} · I_n = A
    /* more patterns... */ }
```

Note that, in the first case, it is imperative to test whether both row and column dimension of the zero matrix and the right-hand argument equal as it is only possible to encode zero matrices of *quadratic* shape.

Transformers

Alternatively, one could also put LMS' transformer infrastructure to work instead of smart constructors. The idea is to unconditionally create IR nodes first and

to analyze the resulting IR afterwards, while looking out for the opportunity to remove unnecessary nodes, possibly in several traversals. This strategy eliminates the need for duplicate dimension checks and was indeed employed in the SpiralS prototype [3, p. 130]. However, in retrospect it was concluded that the usage of "transformations that match and replace a whole subgraph of a given DSL [...] proved to be cumbersome. [...] The LMS transformers are not designed for this use case, as they work in a peephole fashion. Matching on a subgraph, or having conditional rewrites that depend on some state further up in the IR representation inherently is accompanied by large code fragments" [4, pp. 53 sq.]. For this reason, transformers are not considered in this work and the duplication of small code fragments is accepted as the lesser of two evils.

Further Optimizations

The previous example demonstrated how to apply *algebraic simplifications* to SPL code, but the same implementation technique can also be used for other kinds of optimizations. For example, the addition of tridiagonal matrices can be executed pointwise on the entries of corresponding diagonals:

$$\text{Tridiag}_n(a,b,c) + \text{Tridiag}_n(d,e,f) = \text{Tridiag}_n(a+d, b+e, c+f).$$

This equation translates into the following Scala code:

```
override protected def spl_plus(lhs: Exp[SPL], rhs: Exp[SPL]) =
  (lhs, rhs) match {
    case (Const(Tridiag(n, a, b, c)), Const(Tridiag(m, d, e, f)))
      if n == m => Tridiag(n, a+d, b+e, c+f)
    case _ => super.spl_plus(lhs, rhs) }
```

This technique is commonly referred to as *constant folding* [7, p. 68], i.e. turning multiple dynamic constants into a single one. Notice that, even though the matrices themselves are dynamic, the entries on their diagonals are *not* because they are of plain type `Double`, not `Rep[Double]`. Therefore, the results of performing the computations `a + d`, `b + e` and `c + f` are already obtained during generator time and will be inlined as constants in the generated code, which is an example of *precomputation*.

Similar optimization potential can be identified for the tensor product, for example:

$$\mathrm{I}_n \otimes \mathrm{I}_m = \mathrm{I}_{n \cdot m} \quad \text{and} \quad 0_n \otimes B_{n \times n} = 0_{n \cdot m},$$

These rules are encoded as follows:

```
override protected def spl_tensor(lhs: Exp[SPL], rhs: Exp[SPL]) =
  (lhs, rhs) match {
    case (Const(I(n)), Const(I(m))) => I(n * m)
    case (Const(Zero(n)), SPLExp(rows, cols))
```

```
    if rows == cols => Zero(n * rows)
  case _ => super.spl_tensor(lhs, rhs) }
```

Encoding Domain-Specific Knowledge

Attentive readers may rightfully point out that zero and identity matrices are just special kinds of tridiagonal matrices, i.e.

$$0_n = \text{Tridiag}_n(0,0,0) \quad \text{and} \quad \text{I}_n = \text{Tridiag}_n(0,1,0).$$

Yet, a computation like `Tridiag(3, 1, 2, 3) + I(3)` is currently not optimized by `spl_plus`. This is because the pattern extractor `Tridiag` only matches instances of class `Tridiag`, and not of class `I` or `Zero`. To solve this problem, one could add additional patterns in the body of `spl_plus` (and in all other smart constructors, where necessary), such as

```
case (Const(Tridiag(n, a, b, c)), Const(I(m)))
  if n == m => Tridiag(n, a, b + 1, c)
case (Const(Tridiag(n, a, b, c)), Const(Zero(m)))
  if n == m => Tridiag(n, a, b, c)
```

However, there is a less obscuring and more scalable approach, namely modifying the pattern extractor `Tridiag` itself. Just like any other subclass of `SPL`, `Tridiag` is declared as `case class` which instructs the Scala compiler to automatically create a companion object with suitable `apply` and `unapply` methods, amongst others. But it is still possible to define one's own implementation for these methods. We just have to add an object `Tridiag` to the same source file in which class `Tridiag` is defined:

```
object Tridiag {
  def unapply(s: SPL): Option[(Int, Int, Int, Int)] = s match {
    case t: Tridiag => Some(t.n, t.a, t.b, t.c)
    case i: I => Some(i.n, 0, 1, 0)
    case z: Zero => Some(z.n, 0, 0, 0)
    case _ => None } }
```

We just provided the program generator with valuable domain-specific knowledge, therefore allowing it to treat zero and identity matrices as special kinds of tridiagonal matrices. In other words, the pattern extractor `Tridiag` now also matches instances of `I` and `Zero`, e.g.:

```
I(3) match { case Tridiag(n,a,b,c) => (n,a,b,c) } == (3,0,1,0)
Zero(7) match { case Tridiag(n,a,b,c) => (n,a,b,c) } == (7,0,0,0)
```

Therefore, we need not change the implementation of any operators and we achieve the desired optimizations, too. Consequently, the computation

```
Tridiag(3, 1, 2, 3) + I(3)
```

Table 3.4: Optimizing rewrites for SPL expressions. The subscript of A and B denotes the (square) matrix size and s refers to a real number as scalar value. The symbol T_n is used as a shorthand for $\mathrm{Tridiag}_n$.

Operation	Optimizations
Addition	$0_n + B_n = B_n$
	$A_n + 0_n = A_n$
	$\mathrm{T}_n(a,b,c) + \mathrm{T}_n(d,e,f) = \mathrm{T}_n(a+d, b+e, c+f)$
Matrix Multiplication	$\mathrm{I}_n \cdot B_{n\times m} = B_{n\times m}$
	$A_{n\times m} \cdot \mathrm{I}_m = A_{n\times m}$
	$0_n \cdot B_n = 0_n$
	$A_n \cdot 0_n = 0_n$
	$0_n \cdot B_n = 0_n$
	$\mathrm{T}_n(0,a,0) \cdot \mathrm{T}_n(0,b,0) = \mathrm{T}_n(0,ab,0)$
Scalar Multiplication	$0 \cdot B_m = 0_m$
	$1 \cdot B_m = B_m$
	$s \cdot \mathrm{T}_n(a,b,c) = \mathrm{T}_n(sa, sb, sc)$
Tensor Product	$\mathrm{I}_n \otimes \mathrm{I}_m = \mathrm{I}_{nm}$
	$0_n \otimes B_m = 0_{nm}$
	$A_n \otimes 0_m = 0_{nm}$

is now simplified to `Tridiag(3, 1, 3, 3)`. Further custom pattern extractors may be added, in particular for `I` and `Zero`, but their implementation is omitted due to space limitations.

Note that these extractors behave predictably. Internally, the `unapply` method is invoked exactly once per pattern match. Moreover, the decision which pattern matches is made completely deterministic, given that `unapply` is implemented sensibly. Another benefit of extractors is that all domain-specific information is listed in a central location (within the body of `unapply`) and not scattered across the implementation of several operators.

Classic Compiler Optimizations

All other optimizing rewrites, as summarized by Table 3.4, are implemented analogously. On top of that, LMS' general optimizations [7, ch. 11.1], consisting of CSE, dead code elimination (DCE) and code motion, are also enabled without the need for any additional work by the programmer. These are showcased in Section 4.2.

Numerical Challenges

Overall, we resort to a rich set of compiler optimizations. Yet, there is still room for improvement, especially with regard to precomputation or constant folding. Since SPL has been developed with a mathematical domain in mind, it is important to consider the numerical flaws inherent to floating-point arithmetic. Currently, we rely on Scala's `Double` data type to represent real numbers $x \in \mathbb{R}$, which is problematic. For example, the entries on the diagonals of tridiagonal matrices are `Doubles`. Let us assume a mathematician wants to compute

$$\text{Tridiag}_3(0, \sqrt{2}, 0) \cdot \text{Tridiag}_3(0, \sqrt{2}/2, 0).$$

Therefore, an SPL programmer could write the following:

```
Tridiag(3, 0, math.sqrt(2), 0) * Tridiag(3, 0, math.sqrt(2) / 2, 0)
```

Ideally, the program generator replaces this entire line of code with the constant `I(3)` as $\sqrt{2} \cdot \sqrt{2}/2 = 1$ and $\text{Tridiag}_3(0, 1, 0) = \text{I}_3$. But it is unable to do so. A short experiment in the Scala interpreter demonstrates the issue:

```
scala> math.sqrt(2)
res0: Double = 1.4142135623730951
scala> res0 * res0
res1: Double = 2.0000000000000004
scala> res1 / 2
res2: Double = 1.0000000000000002
```

The problem is that

```
math.sqrt(2) * math.sqrt(2) == 2.0000000000000004 != 2
```

Hence, the result is `Tridiag(3, 0, 1.0000000000000002, 0)` and not the actual `I(3)`. Tiny numerical offsets like this one are troublesome since they falsify calculation results ever so slightly, which may accumulate to large errors over time. Also, they prevent potential optimizations from being applied in subsequent computations. For instance, multiplications with the identity matrix are superfluous and can be left out, but this is not the case for said tridiagonal matrix. In summary, it is best not to use `Doubles`.

Instead, we rely on a specialized numeric library called *Spire*[1]. It provides the useful data type `Real`, which implements "computable real numbers" by representing them via "a function from a desired precision to the closest approximate value", which mitigates the numerical problems discussed above, since computations are carried out symbolically, when possible. However, this change also requires us to adapt the current implementation of numerous SPL subclasses, operators and

1 https://github.com/non/spire

IR nodes. `Real` integrates quite well into the Scala language so that changes are restricted to merely replacing the type `Double` with `Real` where necessary, for example

```
case class Tridiag(private val n: Int, a: Real, b: Real, c: Real)
    extends SPL(n) { /* ... */ }
def infix_*>(n: Real, spl: Rep[SPL]): Rep[SPL] = spl_stimes(n, spl)
override protected def spl_stimes(n: Real, spl: Exp[SPL]): Exp[SPL] =
  ScalarTimes(n, spl)
class ScalarTimes private (val n: Real, val spl: Exp[SPL],
    override val rows: Int, override val cols: Int) extends SPLDef
```

And so on. One may also want to define implicit conversions of `Int`s and `Double`s to their `Real` counterparts in trait `SPLOps`:

```
implicit def int2Real(i: Int): Real = Real(i)
implicit def double2Real(d: Double): Real = Real(d)
```

Scala's "native" data types `Int` and `Double` profit from literal value syntax. Implicit conversions enable the same benefits for custom data types, like `Real`, which allows for high convenience and compatibility.

3.5 An Internal Representation of the Programming Language C

Motivation

So far, we have only worked with SPL, a matrix formalism that allows us to express complex computations on a high level of abstraction. As laid out in Section 3.4, this representation is also well suited to apply *high-level* domain-specific optimizations, which helps for performance. But eventually, we would like to execute SPL programs on actual hardware, which usually requires a much *lower level* of abstraction. To bridge this gap, SPL code needs to be translated into a semantically equivalent program that operates on a sufficiently low level of abstraction, i.e. "closer to the machine". Here, the target language of choice is C, which is often considered a "portable assembly language".

Representing C Code

While it is possible to directly unparse the IR produced by an SPL program to C code, this would also mean to deliberately leave out the opportunity of applying general *low-level* optimizations. Due to its high level of abstraction, SPL is unfit for the application of such low-level rewritings. This motivates the definition of a new "intermediate" DSL which we call CIR (C internal representation). In contrast to SPL, this particular DSL is intended as a means of representing code that operates on the comparatively low abstraction level of the general purpose programming

language C. Another crucial difference is that CIR is not designed as user-facing DSL. Users of the generator (such as mathematicians) may program directly in SPL but not in CIR as the latter is meant for use within the program generator only.

Interface

The implementation of CIR is analogous to SPL, but requires less effort. First, an interface trait CIR is created. LMS comes with several ready-made DSL traits[2] from which CIR can be assembled. Each trait thereby represents a certain kind of "functionality" to be made available in CIR client code. In other words, we have to explicitly white list language features by mixing in the corresponding traits. A small subset of C already suffices for our purposes:

(a) complex SPL computations on matrices are expressed in terms of rudimentary arithmetic operations on numerical data types;
(b) SPL vectors have no direct mapping as data type in C, hence the next best and also most natural choice is to use arrays instead.

These considerations lead to the following interface definition:

```
trait CIR extends Base
    with PrimitiveOps   // Basic arithmetic required by NumericOps.
    with NumericOps     // More advanced arithmetic on numeric types.
    with LiftPrimitives // Implicit conversions from Int to Rep[Int] etc.
    with LiftNumeric    // Automatically stages numeric expressions.
    with Variables      // Allows for use of (mutable) variables.
    with LiftVariables  // Automatically stages variables if needed.
    with Effects        // Enables operations with side effects.
    with ArrayOps with SeqOps { // Supports arrays in the generated code.
  def comment(msg: String): Rep[Unit] }
```

As usual, CIR extends LMS' Base trait to bring the Rep type constructor into scope. Next, we only allow certain language features in CIR code by mixing in the appropriate traits, which are all of them already bundled by LMS. In addition, a custom function comment is provided. It creates a comment with the given string msg as content in the generated code, which is helpful for both development and debugging of the DSL. Its result type is Rep[Unit], which corresponds to void in C code. Finally, to make CIR a non-userfacing DSL, we can use a Scala feature known as "Scopes of Protection" [11, pp. 288–290] which allows for fine-grained adjustment of visibility rules. The access modifiers private and protected can be augmented with an additional identifier, i.a. denoting an enclosing package. Assuming that the program generator is placed in the package spirals, we can

2 These traits are most often utilized to represent staged Scala code because LMS has been developed as MSP framework for Scala, after all. But since the process of staging code is decoupled from the act of generating code, the traits are general enough to model other target languages as well. When unparsing, one just has to use C syntax instead of Scala syntax.

restrict the visibility of CIR to the program generator by making the DSL "package private":

```
private[spirals] trait CIR extends ... // the rest as before
```

Unlike Java, Scala assumes the default visibility (that is, when no access modifier is explicitly given) to be public. The syntax `private[spirals]` makes CIR accessible only within the package `spirals`.

Integration of Real

What was not mentioned yet is that a fair bit of boilerplate code is necessary to enable support for Spire's `Real` data type in CIR code. The good news is that traits like `NumericOps` do not hard code their underlying representation of numbers to any particular data type, but rather rely on the general interface defined by Scala's `Numeric` type class. Hence, making `Real` an instance of `Numeric`[3] already suffices to make it available for use in CIR:

```
implicit object RealIsNumeric extends Numeric[Real] {
  override def plus(x: Real, y: Real): Real = x + y
  override def times(x: Real, y: Real): Real = x * y
  override def negate(x: Real): Real = -x
  override def fromInt(x: Int): Real = Real(x)
  override def compare(x: Real, y: Real): Int = x compare y
  /* further methods not shown due to space limitations */ }
```

Implementation

The next step is to give a concrete implementation of CIR in trait CIRExp:

```
private[spirals] trait CIRExp extends CIR with BaseExp
    with PrimitiveOpsExp with NumericOpsExp
    with VariablesExp    with EffectExp
    with ArrayOpsExp     with SeqOpsExp {
  case class Comment(msg: String) extends Def[Unit]
  override def comment(msg: String): Exp[Unit] =
    reflectEffect(Comment(msg)) }
```

Again, this is achieved simply by mixing in the corresponding implementation traits, which we already have at our disposal in LMS' core library. The only real work that remains is to define the `comment` function. Therefore, a suitable IR node `Comment` is introduced, which holds the content of the comment as `String` and, since it represents a composite operation, extends `Def[Unit]`. Consequently, `comment` yields a new instance of this IR node. However, it is important to observe that `reflectEffect` is used instead of `toAtom` to return an atomic value. The reason is that `comment` introduces a *side effect* in CIR programs, that is, it

3 Curiously enough, Spire defines its own type class for numeric data types which also happens to be called `Numeric`, but refuses to provide an appropriate instantiation of `Real` for Scala's own `Numeric` type class.

doesn't produce code that yields an interesting result. It merely causes a string (a comment) to be inserted into the generated code.

Handling Side Effects

The function `reflectEffect` is defined in trait `Effects`. Its signature is identical to `toAtom` (c.f. Page 14):

```
def reflectEffect[T](d: Def[T]): Exp[T]
```

Until now, we have only considered purely functional DSL programs. But when side effects come into play, special attention must be paid. For instance, remember that `toAtom` eliminates duplicate code. If operations have side effects, this optimization could alter the intended meaning of the DSL program. Furthermore, LMS can no longer safely apply code motion, i.e. changing the order of statements in the object program to improve reuse of information, cache locality, etc. Thus, LMS requires the programmer to track side effects via `reflectEffect`, which allows the framework to reason about the impact that DSL operations can have [7, ch. 9.4]. For example, `reflectEffect` causes that effectful operations are assigned a fresh new symbol in any case.

Optimizations

Finally, an optimized implementation `CIRExpOpt` is put together. Realize that these are precisely the optimizations that motivated the introduction of CIR in the first place. The definition of `CIRExpOpt` shows a recurring picture; composing several preexisting LMS traits is all it takes:

```
private[spirals] trait CIRExpOpt extends CIRExp
  with PrimitiveOpsExpOpt with NumericOpsExpOpt
  with VariablesExpOpt    with ArrayOpsExpOpt
```

`PrimitiveOpsExpOpt` and `NumericOpsExpOpt` accomplish *algebraic simplification* and *constant folding* with regard to addition, subtraction, multiplication and division for numeric data types. These optimizations do not only apply for native data types such as `Int` and `Double`, but also for Spire's `Real` type, since we have made it an instance of Scala's `Numeric` type class. Said optimizations are again realized using smart constructors, which were already shown extensively in Section 3.4. For example, `NumericOpsExpOpt` defines addition as follows:

```
override def numeric_plus[T: Numeric](a: Exp[T], b: Exp[T]): Exp[T] = {
  val num = implicitly[Numeric[T]]
  (a, b) match {
    // constant folding
    case (Const(x), Const(y)) => Const(num.plus(x, y))
    // algebraic simplification
    case (Const(x), y) if x == num.zero => y // 0 + y = y
    case (x, Const(y)) if y == num.zero => x // x + 0 = x
    // all other cases
    case _ => super.numeric_plus(a, b) } }
```

Furthermore, `VariablesExpOpt` eliminates redundancy in reads and writes of variables, e.g. by copy propagation [7, p. 79], and `ArrayOpsExpOpt` achieves the same for arrays in particular. Moreover, LMS' general optimizations (c.f. Page 36) are also applied. They do not require any refined IR semantics and generally work without the need of mixing in any optimization traits at all.

Section 3.6 covers the translation from SPL to CIR. During this process, all domain-specific knowledge captured by SPL will be translated away, too. Hence, it was a good idea to perform domain-specific optimizations directly on SPL code. Had we decided to implement them only now, the low abstraction level of CIR would have made it much more difficult to achieve them.

3.6 Translation from SPL to CIR

As mentioned earlier, the translation from SPL to CIR is a fundamental step. The challenge is to boil down complex SPL matrix algorithms to CIR programs that make use of only rudimentary arithmetic operations manipulating scalar values. In order to bridge this gap, the key insight is that every SPL matrix M can be represented as a CIR *function* that takes a vector x, performs the computation $y := M \cdot x$ and returns the resulting vector y. Table 3.5 outlines how to translate M into basic sequential loop code.

Caveats

Before we turn to the implementation, there are a few preliminary considerations to make. As stated in Section 3.5, vectors do not have a direct representation as data type in C. Hence, the next best solution is to use arrays. But in C it is very problematic to return arrays, or more accurately, pointers since the array in question might be a local variable. It is important to realize that all local stack frames are popped once control flow runs out of the current scope, such as the body of a function. For this reason, the returned pointer would refer to a location in memory that has an *undefined* state. The problem could be avoided if memory for the array is allocated on the heap rather than on the stack. But this solution is taxed with heavy runtime overhead. A better approach is to let callers of the function preallocate the required amount of memory for us. Therefore, we introduce an additional formal parameter, namely a pointer that represents the array to be yielded, and change the function's return type to `Unit` (i.e. `void` in C).

Groundwork

We provide a trait `SPL2CIR` that contains the facilities to translate from SPL to CIR. During the translation process, it is mandatory to examine the internal structure of a given SPL program. Hence, we demand that users of `SPL2CIR` supply an appropriate instance of `SPLOpsExp`. In order to produce CIR code, it suffices to gain access to the interface `CIR`, which must once again be given by users.

Table 3.5: Translation of matrix formulas into Matlab-style pseudo code. The subscript of matrices A and B specifies the (square) matrix size. The expression `x[b:s:e]` denotes the subvector of `x`, starting at base `b`, ending at index `e` and extracted at stride `s`. The table has been taken from [1, p. 3].

Matrix formula	Matlab-style pseudo code
$y = \mathrm{I}_n\, x$	`for (i = 0; i < n; i++)` `  y[i] = x[i];`
$y = 0_n\, x$	`for (i = 0; i < n; i++)` `  y[i] = 0.0;`
$y = \mathrm{G}^{n\times N}_{b,s}\, x$	`for (i = 0; i < n; i++)` `  y[i] = x[b+i*s];`

$y = \mathrm{S}^{N\times n}_{b,s}\, x$	`y = 0.0;`
	`for (i = 0; i < n; i++)`
	`  y[b+i*s] = x[i];`

$y = \mathrm{Tridiag}_n(a,b,c)\, x$	`y[0] = b * x[0] + c * x[1];`
	`for (i = 1; i < n-1; i++)`
	`  y[i] = a * x[i-1] + b * x[i] + c * x[i+1];`
	`y[n-1] = a * x[n-2] + b * x[n-1];`

$y = (A_n + B_n)\, x$	`y[0 : 1 : n-1] = A(x[0 : 1 : n-1]) +` `  B(x[0 : 1 : n-1]);`
$y = (A_n \cdot B_n)\, x$	`t[0 : 1 : n-1] = B(x[0 : 1 : n-1]);` `y[0 : 1 : n-1] = A(t[0 : 1 : n-1]);`
$y = [A_n \mid B_n]\, x$	`y[0 : 1 : n-1] = A(x[0 : 1 : n-1]) +` `  B(x[0 : 1 : n-1]);`
$y = \begin{bmatrix} A_n \\ B_n \end{bmatrix} x$	`y[0 : 1 : n-1]   = A(x[0 : 1 : n-1]);` `y[n : 1 : 2*n-1] = B(x[0 : 1 : n-1]);`

$y = \left(\prod_{i=0}^{k-1} A_i\right) x$	`y = x;`
	`for (i = 0; i < k; i++) {`
	`  x = y; y = A(i, x); }`

$y = (\mathrm{I}_m \otimes A_n)\, x$	`for (i = 0; i < m; i++)`
	`  y[i*n : 1 : i*n+n-1] =`
	`  A(x[i*n : 1 : i*n+n-1]);`

$y = (A_m \otimes \mathrm{I}_n)\, x$	`for (i = 0; i < n; i++)` `  y[i : n : i+m*n-n] = A(x[i : n : i+m*n-n]);`

```
trait SPL2CIR {
  val source: SPLOpsExp
  val target: CIR
  ... }
```

The members `source` and `target` represent source and target languages for the translation. Both are merely abstract at this point. They need to be concretely defined when trait `SPL2CIR` is instantiated. Since we are now working with two DSLs, it is often necessary to explicitly refer to one DSL or the other in order to prevent ambiguities. This is done via the `source` or `target` identifier, and path-dependent types. For instance, the `Rep` type constructor occurs in both DSLs, hence it is convenient to define short aliases, like so:

```
type SRep = source.Rep[SPL]
type TRep[T] = target.Rep[T]
```

A Translation Function

Thereafter, we begin with the implementation of an actual translation function, which we call `translate`. It takes a staged SPL expression, i.e. of type `SRep`, and yields another function representing the corresponding CIR code. In turn, this particular function then requires two arrays and produces no interesting result, that is of type `Unit`. It is vital to highlight that there is a crucial difference between the types `Rep[A => B]` and `Rep[A] => Rep[B]`. The former represents a staged function from some type `A` to another type `B`. Therefore, there will be a function call in the generated code. In contrast, the latter stands for a static function that operates on dynamic values of type `Rep[A]` and `Rep[B]`. In the generated code, its body will be inlined. This is the desired behavior, so the proper return type for `translate` is

```
(TRep[Array[Real]], TRep[Array[Real]]) => TRep[Unit]
```

Take note that the function itself is not staged, only its parameters and result are. Also, `TRep[_]` must now be used instead of `SRep`. Putting it all together, we obtain the following signature for `translate`:

```
def translate(x: SRep):
    (TRep[Array[Real]], TRep[Array[Real]]) => TRep[Unit]
```

Since Scala is a full-fledged functional programming language, we can use a lambda expression in the body of `translate` to create an anonymous function to be returned:

```
def translate(x: SRep):
    (TRep[Array[Real]], TRep[Array[Real]]) => TRep[Unit] =
  (in, out) => ...
```

The syntax `(in, out) => ...` denotes the lambda expression. Its two formal parameters are an input array `in` and output array `out`. Their respective type

`TRep[Array[Real]]` is already inferred from `translate`'s signature. Inside the lambda's body, we need to inspect the structure of the given SPL expression `x` in a pattern match and react accordingly:

```
(in, out) => x match { ... /* case distinctions are put here */ }
```

3.6.1 Atomic Expressions

Zero and Identity Matrix

Finding suitable CIR functions is simple for atomic SPL expressions. The easiest cases are $M = 0_n$ and $M = \mathrm{I}_n$. In the former one, multiplying 0_n with a given input vector x of size n results in the zero vector of size n. In the latter case, the result of $\mathrm{I}_n \cdot x$ is just x. As suggested by Table 3.5, we can use a for-loop to iterate over the output array and set every of its entries to zero, or copy the values from the input array, respectively:

```
case Const(Zero(n)) => for (i <- 0 until n) out(i) = 0
case Const(I(n))    => for (i <- 0 until n) out(i) = in(i)
```

Notice how CIR reuses Scala's syntax and language features, such as the for-loop. Both the case distinctions in the pattern match and the for-loops are static constructs, which means they only appear during the present stage and will have been translated away before the second stage takes place. But the assignments to the output array `out` are dynamic as it is of type `TRep[Array[Real]]`. Consequently, each loop will be unrolled in the generated code: for each iteration step, the loop's body is placed once in the object program. Thereby, the loop variable `i` is replaced with the value it attained during the corresponding iteration. Conceptually:

```
// Zero matrix:                  // Identity matrix:
out(0) = 0                       out(0) = in(0)
out(1) = 0                       out(1) = in(1)
...                              ...
out(n - 2) = 0                   out(n - 2) = in(n - 2)
out(n - 1) = 0                   out(n - 1) = in(n - 1)
```

Tridiagonal Matrix

Remembering Section 3.4, we find that 0_n and I_n are just special kinds of tridiagonal matrices. Hence, a general implementation for $\mathrm{Tridiag}_n(a, b, c)$ is sufficient as it also covers aforementioned special cases. Multiplying $\mathrm{Tridiag}_n(a, b, c)$ with an n-dimensional vector x results in the following:

$$\begin{bmatrix} b & c & & & \\ a & b & c & & \\ & \ddots & \ddots & \ddots & \\ & & a & b & c \\ & & & a & b \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-2} \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b \cdot x_0 + c \cdot x_1 \\ a \cdot x_0 + b \cdot x_1 + c \cdot x_2 \\ \vdots \\ a \cdot x_{n-3} + b \cdot x_{n-2} + c \cdot x_{n-1} \\ a \cdot x_{n-2} + b \cdot x_{n-1} \end{bmatrix}.$$

An appropriate implementation is given below:

```
case Const(Tridiag(n, a, b, c)) =>
  out(0) = b * in(0) + c * in(1)
  for (i <- 1 until n - 1)
    out(i) = a * in(i - 1) + b * in(i) + c * in(i + 1)
  out(n - 1) = a * in(n - 2) + b * in(n - 1)
```

Computing the resulting components for the output vector is handled via a for-loop. But the first and last component require special treatment as there are only two multiplications and one addition to execute instead of three multiplications and two additions. This is done in distinct statements outside of the loop. Even though the constants `a`, `b` and `c` are static, the right-hand side of each assignment also involves reads from the dynamic input array `in`. For this reason, the entire right-hand side can only be evaluated dynamically. The generated code has the structure shown below:

```
out(0) = b * in(0) + c * in(1)
out(1) = a * in(0) + b * in(1) + c * in(2)
...
out(n - 2) = a * in(n - 3) + b * in(n - 2) + c * in(n - 1)
out(n - 1) = a * in(n - 2) + b * in(n - 1)
```

The loop has been unrolled and the iteration variable is inlined as a constant. Note that the optimizations defined in `CIRExpOpt` still apply. As a consequence, the generated code for `Zero(n)` and `I(n)` is heavily simplified; in fact, it is identical to the one created by the hand-crafted implementations presented at the beginning of Section 3.6.1.

Gather Matrix

As a last example, let us consider the gather matrix $G_{b,s}^{n\times m}$. It extracts the entries from an m-dimensional vector, starting at base b with stride s and yields an n-dimensional vector:

```
case Const(G(rows, _, base, stride)) =>
  for (i <- 0 until rows) out(i) = in(base + i * stride)
```

Just like before, this will result in unrolled code. For instance, `G(3, 7, 1, 2)` gives the following:

```
out(0) = in(1)
out(1) = in(3)
out(2) = in(5)
```

Since the variables `base`, `i` and `stride` are all static, the computation `base + i * stride` is performed statically, too, and its result is inlined as a constant. Note that there is no way of enforcing or even testing whether the lengths of the input array `in` and output array `out` actually conform to m or n, respectively. This is

because we relied on callers to create these arrays for us. But at least we can be sure that the correctness of SPL expressions with regard to their dimensions is ensured by the operators defined in trait SPLOpsExp. That is, it is impossible to encounter an ill-formed SPL expression because then it would not have been created in the first place. In CIR code, similar correctness guarantees do *not* exist for input and output arrays.

3.6.2 Composite Expressions

Matrix Multiplication

However, the translation becomes more difficult if M is composite. For instance, let $M = A \cdot B$ where A and B are atomic. As demonstrated before, it is easy to find appropriate functions f and g for A and B, respectively. But it is very hard to translate their product as a whole. In this particular case, the remedy is to harness the associativity of matrix multiplication:

$$(A \cdot B) \cdot x = A \cdot \underbrace{(B \cdot x)}_{=:z} = A \cdot z = y.$$

By rewriting the equation, the composite operation $(A \cdot B) \cdot x$ is broken down into two calculations $z = B \cdot x$ and $y = A \cdot z$ containing only atomic SPL expressions. Notice that the output vector z of the first calculation serves as the input vector for the second one. That is, composing f and g yields the desired function corresponding to matrix M:

$$(f \circ g)(x) = f(\underbrace{g(x)}_{=z}) = f(z) = y.$$

If A or B are composite themselves, it is necessary to repeat this process recursively until the equations only consist of atomic values. Apparently, this strategy also changes the order of evaluation and therefore comes at the cost of having to introduce additional storage for an intermediate result z. This fact is also reflected in the implementation:

```
case Def(Times(lhs, rhs @ SPLExp(rowsR, _), _, _)) =>
  val (l, r) = (translate(lhs), translate(rhs))
  val tmp = NewArray[Real](rowsR)
  r(in, tmp)
  l(tmp, out)
```

Since we want to inspect an SPL fragment that models a composite operation, the pattern extractor Def needs to be used instead of Const. Next, we make use of the custom extractors defined in Sections 3.2 and 3.4 for fine-grained deconstruction of the given IR node. Matrix multiplication as represented by Times takes two

operands `lhs` and `rhs`. The right-hand argument's row dimension determines the size of the intermediate storage. Thus, we need to further inspect `rhs`. For this purpose, it suffices to use the extractor `SPLExp`. To avoid nesting another pattern match within the present one, Scala offers *variable bindings* [11, p. 320] as an alternative. The expression `rhs @ SPLExp(rowsR, _)` binds the right-hand argument of the multiplication to the name `rhs` and also decomposes it at the same time. Its number of rows can now be accessed via the identifier `rowsR`. Now that the given SPL expression is fully analyzed, we can translate `lhs` and `rhs` to obtain the corresponding functions `l` and `r`. Afterwards, a temporary array `tmp` is created, which will be used to store the intermediate result z. The command `NewArray` defined by trait `ArrayOps` causes that the construction of the array is delayed until the second program stage is executed. Next, function `r` reads the values from `in` and stores the results in `tmp`, which then in turn serves as the input for `l`. This function writes its own results into the final output array `out`, which also happens to be the result of the entire matrix multiplication.

Matrix Addition

For matrix addition, we resort to the same implementation techniques. Using the law of distributivity, we obtain:

$$(A+B)\cdot x = \underbrace{(A\cdot x)}_{=:y} + \underbrace{(B\cdot x)}_{=:z} = y+z.$$

Vector x is multiplied with both A and B. This produces two intermediate results y and z, which are added together to calculate the final result:

```
case Def(Plus(lhs, rhs, rows, _)) =>
  val (l, r) = (translate(lhs), translate(rhs))
  val resR = NewArray[Real](rows)
  val resL = NewArray[Real](rows)
  r(in, resR)
  l(in, resL)
  for (i <- 0 until rows) out(i) = resL(i) + resR(i)
```

Employing a for-loop, we simultaneously iterate over both intermediate arrays `resR` and `resL`, compute the sum of every component and write the result to the output array `out`.

Tensor Product

The last example covers the tensor product of the identity matrix I_n with another arbitrary matrix M of size $p \times q$:

$$(\mathrm{I}_n \otimes M)\cdot x = \begin{bmatrix} M & & & \\ & M & & \\ & & \ddots & \\ & & & M \end{bmatrix} \begin{bmatrix} \overline{x}_0 \\ \overline{x}_1 \\ \vdots \\ \overline{x}_{n-2} \\ \overline{x}_{n-1} \end{bmatrix} = \begin{bmatrix} M\cdot\overline{x}_0 \\ M\cdot\overline{x}_1 \\ \vdots \\ M\cdot\overline{x}_{n-2} \\ M\cdot\overline{x}_{n-1} \end{bmatrix}.$$

The notation $\overline{x}_k$ stands for chunks of the input vector x; more precisely, it represents the sub-vector of dimension q, starting at index kq and ending at index $(k+1)\cdot q-1$. The result of $M\cdot\overline{x}_k$ is a p-dimensional vector. The appropriate implementation of $\mathrm{I}_n\otimes M$ is as follows:

```
case Def(Tensor(Const(I(n)), rhs @ SPLExp(rowsR, colsR), _, _)) =>
  val f = translate(rhs)
  for (k <- 0 until n) {
    val (startIn, startOut) = (k * colsR, k * rowsR)
    val (endIn, endOut) = (startIn + colsR, startOut + rowsR)
    val subIn = in.slice(startIn, endIn)
    val subOut = out.slice(startOut, endOut)
    f(subIn, subOut) }
```

The pattern only matches if the left-hand argument to the tensor is the identity matrix. After that, it suffices to translate the right-hand side `rhs` once, since it is duplicated `n`-times. This yields the corresponding function `f`. Next, a for-loop ranging from `0` to `n - 1` is used to perform the computations $M\cdot\overline{x}_k$. Within the loop's body, the starting and ending indices of the input and output sub-arrays are determined with regard to the current loop counter `k`. As usual, the loop will be unrolled and all static variables like the upper and lower index bounds we just calculated will be inlined as constants. After that, we directly take suitable chunks of the input and output arrays (`subIn` and `subOut`) using the function `slice`, which is provided by trait `ArrayOps`, and pass them to `f`. It is important to mention that `slice` demands a lower bound as first parameter, which is inclusive, and an upper bound as second parameter, which is *exclusive*.

Unknown Expressions

These examples should have given a good insight to the translation process. Finally, to handle SPL expressions for which it is not known how to translate them, we include the following as *last* pattern:

```
case _ => throw new GenerationFailedException
```

The wildcard matches any given input and a `GenerationFailedException` is thrown to indicate that the translation failed.

Testing and Debugging

During development, it may be useful to know from which SPL matrix certain CIR statements were created. For this purpose, the `comment` function comes in handy. Putting it to work as laid out below causes that explanatory comments are generated immediately before and after the corresponding CIR code:

```
case Const(I(n)) =>
  comment(s"begin I($n)")
  for (i <- 0 until n) out(i) = in(i)
  comment(s"end I($n)")
```

When preceding the opening quote of a string, Scala interprets the prefix `s` as string interpolation operator [11, ch. 5.3]. It causes that embedded expressions marked with the dollar sign (`$`) are evaluated and the string representation of the result is injected into the processed string. Hence, string interpolation commonly serves as readable and concise alternative to string concatenation:

```
s"begin I($n)"
"begin I(" + n.toString + ")"
```

The expression in the first line uses a processed string literal, while the one in the second line utilizes string concatenation and a call to `toString`. Both expressions are equal, though. Besides that, string interpolation can be useful for other applications as well, such as unparsing CIR code to actual C code as demonstrated in Section 3.8. In order not to clutter the implementation with too many details, the `comment` function is no longer mentioned from now on.

3.6.3 Different Code Styles and Data Representations

Realize that the implementation of `translate` commits to a specific code style and data representation, namely unrolled loop code with precomputation and the use of arrays in the generated program. Essentially, this behavior can be altered by strategically changing the types of certain expressions in CIR code.

Conditional Loop Unrolling

Currently, all loops are unrolled unconditionally, which may be undesired if a loop comprises a large number of iterations. Therefore, one may want enable the generation of loops by explicitly mixing in the traits `Loops` and `RangeOps` to `CIR`. Then, the key is to iterate over *staged* `Range`[4] objects, that is

```
for (i <- (0 until n): Rep[Range]) ...
```

instead of

```
for (i <- (0 until n): Range) ...
```

As outlined in Section 2.3, such for-comprehensions are just syntactic sugar for calls to the `foreach` combinator (among others):

```
for (i <- 0 until n) f(i)  ==  (0 until n) foreach f
```

This assumes that function `f` has the type `Int => Unit`. With a bit of Scala wizardry[5], we can use for-comprehensions to inline loops based on a static condition. For that, we define an auxiliary function `unrollIf`:

4 In Scala, a `Range`, as created by the expression `0 until n`, represents an integer-valued interval ranging from `0` until `n`, exclusively, with a step size of `1`. To illustrate: `(0 until 5).toList == List(0, 1, 2, 3, 4)`.

5 Credits to Tiark Rompf: `https://scala-lms.github.io/tutorials/shonan.html`

```
def unrollIf(c: Boolean)(r: Range) = new {
  def foreach(f: Rep[Int] => Rep[Unit]): Rep[Unit] = {
    if (c) for (j <- (r.start until r.end): Range)      f(j)
    else   for (j <- (r.start until r.end): Rep[Range]) f(j) } }
```

It takes a boolean `c` to decide whether the loop should be unrolled, and a range `r` over which to iterate. Then, `unrollIf` returns an instance of an anonymous class which contains a suitable definition of `foreach`. If `c` evaluates to `true`, we simply iterate over the given static range, which unwinds the loop. Otherwise, `r` is turned into a *dynamic* range, which preserves the loop in the object program. This makes `unrollIf` helpful for use within the `translation` function, for example like so:

```
case Const(I(n)) => for (i <- unrollIf(n <= 7)(0 until n)) out(i) = in(i)
```

For "small" identity matrices, i.e. with a dimension `n` less than or equal to the arbitrarily chosen 7, the loop is unrolled. This way, the overhead of evaluating the loop condition is avoided in the generated code, which may improve performance. However, this benefit is reversed for larger values of `n`. As code size grows linearly with `n`, the instruction cache will be exhausted eventually once `n` becomes too big. In this case, a loop is more efficient.

Current Limitations

However, program generation is currently limited to unrolled code as the present set of breakdown rules (c.f. Table 2.1) is unfit for the generation of optimized loops and recursive code. Adding the missing loop-level optimizations is an ongoing research effort that requires further extensions of the code generation and optimization engines to support the code patterns seen in multigrid solvers [1, p. 9].

Precomputation

Furthermore, precomputation can be disabled simply by lifting the appropriate variables, therefore delaying their evaluation until the next program stage takes place.

Scalar Replacement

Finally, it is possible to replace computations involving arrays with ones that operate on scalar variables, an optimization technique referred to as *scalar replacement*. Recall that the use of `NewArray[T](n)` causes the creation of dynamic arrays (i.e. of type `Rep[Array[T]]`) with length `n`. In contrast, constructing static arrays via `new Array[Rep[T]](n)` results in the generation of `n` distinct "scalar" variables of type `Rep[T]`. Abstracting over the choice which data representations and code patterns to use is possible by abstracting over staging decisions. Achieving this goal with help of type classes has been studied by Ofenbeck et al. [3] and was subject to extended academic research by Ofenbeck [4] at the time of this writing. However, a discussion about selective staging is beyond the scope of this work.

3.7 Stencil Computations

3.7.1 Current Limitations

It has been stated many times that I_n and 0_n are specializations of $\mathrm{Tridiag}_n(a,b,c)$. Likewise, $\mathrm{Tridiag}_n(a,b,c)$ is a special kind of *Toeplitz* matrix, that is, each descending diagonal from left to right consists of the same constant values. We will only consider *quadratic* Toeplitz matrices in this work. The breakdown rules in Table 2.1 often resolve to the tensor product of Toeplitz matrices, which has an interesting structure. In Section 3.6, we have already seen that

$$\mathrm{I}_n \otimes T = \mathrm{Tridiag}_n(0,1,0) \otimes T = \begin{bmatrix} T & 0_n & & & \\ 0_n & T & 0_n & & \\ & \ddots & \ddots & \ddots & \\ & & 0_n & T & 0_n \\ & & & 0_n & T \end{bmatrix},$$

where $T = T_{n \times n}$ is Toeplitz. Furthermore, let $M := \mathrm{Tridiag}_n(a,b,c)$, $A := a \cdot M$, $B := b \cdot M$ and $C := c \cdot M$. Then,

$$M \otimes M = \begin{bmatrix} B & C & & & \\ A & B & C & & \\ & \ddots & \ddots & \ddots & \\ & & A & B & C \\ & & & A & B \end{bmatrix}.$$

In other words, the tensor product of tridiagonal matrices yields yet another "tridiagonal" matrix. But this time, the entries on the diagonals are matrices themselves (2-dimensional entities) instead of scalar values (1-dimensional entities). The same holds for the tensor product of general Toeplitz matrices.

However, the translation into CIR code is currently unwieldy. For instance, $M \otimes M$ requires specialized code involving nested for-loops: an outer loop to iterate over the structure of the "outer" tridiagonal matrix, and an inner loop to iterate over the "inner" tridiagonal matrices on the diagonals. Due to space limitations, only an excerpt of the implementation is shown:

```
// tridiagonal matrices B and C on the first row
out(0) = ...
for (i <- 1 until n - 1) out(i) = ...
out(n - 1) = ...
```

```
// tridiagonal matrices A, B and C on the middle rows
for (i <- 1 until n - 1) {
  out(i * n) = ...
  for (j <- 1 until n - 1) {
    out(i * n + j) =
      aa * in((i - 1) * n + j - 1) +
      ab * in((i - 1) * n + j    ) +
      ac * in((i - 1) * n + j + 1) +
      ba * in( i      * n + j - 1) +
      bb * in( i      * n + j    ) +
      bc * in( i      * n + j + 1) +
      ca * in((i + 1) * n + j - 1) +
      cb * in((i + 1) * n + j    ) +
      cc * in((i + 1) * n + j + 1) }
  out((i + 1) * n - 1) = ... }

// tridiagonal matrices A and B on the last row
out(nn - n) = ...
for (i <- 1 until n - 1) out(nn - n + i) = ...
out(nn - 1) = ...
```

What makes this solution suboptimal is that the implementation for ordinary tridiagonal matrices $\text{Tridiag}_n(a, b, c)$ is not reused. Instead, it is duplicated several times and slightly modified on every occasion. What's more, the implementations for $M \otimes M$ and $\text{I}_n \otimes M$ (c.f. Section 3.6) are completely different, even though both of them are merely special parameterizations of the same kind of computational problem, namely finding the tensor product of two Toeplitz matrices.

To overcome these issues, we have to restructure our code. In particular, we have to change the encoding of Toeplitz matrices and tensor products of such matrices. In CIR code, their implementations always follow the same pattern, which is well represented by so called stencil computations.

3.7.2 Fundamentals and Terminology

A *stencil*[6] is a numerical kernel (i.e. a function) that captures the essence of a recurring computational pattern, hence the name. Revisiting the introduction to multigrid methods given in Section 2.1.2, we limit ourselves to stencils that operate on a discretized finite 1- or 2-dimensional grid of variables. Each variable (i.e. point in the grid) has the same relative relationship to its neighbors. The concrete shape of this neighborhood is determined by the stencil. Next, the value

6 Also referred to as *filter*, particularly in the domain of signal processing.

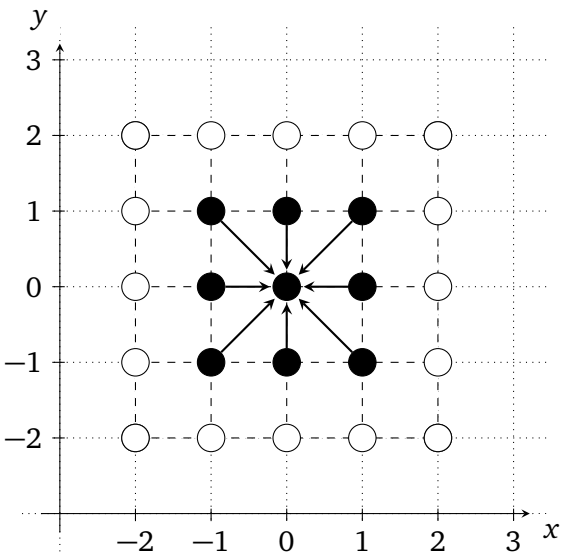

Figure 3.4: Depiction of a 2-dimensional 9-point stencil operating on a 5×5 square grid of variables. Each circle represents a variable on the grid. The stencil is currently applied to $(0, 0)$. The arrows denote the relationship of neighboring points (drawn as solid black circles) to the point $(0, 0)$. In contrast, there are no data dependencies between the white circles and the point $(0, 0)$.

of each variable is obtained by applying the numerical kernel to it. Thereby, the neighboring points of the current variable are usually taken into account as well. In this context, a stencil involving p points is called a p-point stencil. Figure 3.4 illustrates a 9-point stencil operating on a 5×5 grid. Furthermore, the stencil may weigh neighboring points differently, i.e. certain variables may have a greater impact on the computation than others.

In particular, problems arise when points on the boundaries of the grid are dealt with; some of the neighbors referred to may not exist as they are located outside of the grid. Thus, we have to define how such neighbors should be handled, or in other words, we have to specify a boundary condition. In the Dirichlet case, it is common to simply eliminate said points, i.e. we assume a constant value of 0 for every point outside of the grid.

Applying the kernel function successively to each lattice point is called *stencil computation*[7]. Such computations may require several iterations, i.e. repeated applications of the stencil for every point in the grid.

3.7.3 Toeplitz Matrices

Mathematical Examination

The distinctive property of Toeplitz matrices is that each diagonal contains the same entries. Formally, given a square matrix $T = [t_{i,j}]_{0 \leq i,j < n}$, the equation $t_{i,j} = t_{i+1,j+1}$ holds. Therefore, T is fully characterized by (a) its size $n \times n$; and

7 Mathematically speaking, we implement the *convolution* of a filter.

(b) a list of $2n-1$ diagonal entries, also called the *coefficient list*. It contains the entries

$$t_{n-1,0},\ t_{n-2,0},\ \ldots,\ t_{1,0},\ t_{0,0},\ t_{0,1},\ \ldots,\ t_{0,n-2},\ t_{0,n-1}$$

in that specific order. Note that the coefficient list is always odd-numbered. Thus, we can well-define the *center* of such lists as $t_{0,0}$. For example, the 3×3 Toeplitz matrix

$$M := \begin{bmatrix} c & d & e \\ b & c & d \\ a & b & c \end{bmatrix}$$

has the coefficient list a, b, c, d, e. Its center is c. Now, let us consider the product of M with a 3-dimensional vector x:

$$Mx = \begin{bmatrix} c & d & e \\ b & c & d \\ a & b & c \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} c \cdot x_0 + d \cdot x_1 + e \cdot x_2 \\ b \cdot x_0 + c \cdot x_1 + d \cdot x_2 \\ a \cdot x_0 + b \cdot x_1 + c \cdot x_2 \end{bmatrix} =: \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}.$$

We represent M's coefficient list as a row vector

$$c_M := \begin{bmatrix} a & b & \mathbf{c} & d & e \end{bmatrix},$$

with center c printed in boldface for emphasis. We observe that

$$y_0 = c_M \cdot \begin{bmatrix} 0 \\ 0 \\ \mathbf{x_0} \\ x_1 \\ x_2 \end{bmatrix}, \quad y_1 = c_M \cdot \begin{bmatrix} 0 \\ x_0 \\ \mathbf{x_1} \\ x_2 \\ 0 \end{bmatrix}, \quad y_2 = c_M \cdot \begin{bmatrix} x_0 \\ x_1 \\ \mathbf{x_2} \\ 0 \\ 0 \end{bmatrix}.$$

That is, Mx can be interpreted as 3-point stencil computation with Dirichlet boundary condition. Hereby, the input vector x serves as 1-dimensional 3×1 grid from which the kernel function reads. It takes the entries in c_M, multiplies them with the points in the grid, adds up the results to yield the new value of the current point and writes the outcome to y . Note that the grid is always arranged in such a way that the stencil's center c is multiplied with the currently considered point. For instance, y_0 is the result of applying the stencil to x_0.

Scala Representation

To implement stencil computations in the code generator, we drop the case classes `I`, `Zero` and `Tridiag` in favor of the more general `Toeplitz1`. It represents an $n \times n$ Toeplitz matrix via a coefficient list `coeffs`:

```
case class Toeplitz1(n: Int, coeffs: List[Real]) extends SPL(n) {
  require(coeffs.length == 2 * n - 1)
```

```
  val center: Int = coeffs.length / 2 }
```

It is ensured that the length of `coeffs` is valid with regard to the given dimension `n`. The center of the coefficient list is given by the constant `center`. Note that a matrix $\text{Tridiag}_3(1,2,3)$ must now be created as follows:

```
Toeplitz1(3, List(0, 1, 2, 3, 0))
```

This raises eligible doubts regarding ease of use and compatibility to previously written SPL code. But in most cases, these concerns can be sorted out by defining a standalone Scala object `Tridiag` containing appropriate implementations of an `apply` factory and `unapply` extractor method:

```
object Tridiag {
  def apply(n: Int, a: Real, b: Real, c: Real): SPL = {
    require(n > 1)
    val zeros = List.fill(n - 2)(Real.zero)
    val coeffs = zeros ++ List(a, b, c) ++ zeros
    Toeplitz1(n, coeffs) }
  def unapply(arg: SPL): Option[(Int, Real, Real, Real)] = ...  }
```

The `apply` function has the same signature as the constructor of the former case class `Tridiag`. It automatically builds the correct coefficient list from the given arguments and returns a fitting instance of `Toeplitz1`. Therefore, the matrix $\text{Tridiag}_3(1,2,3)$ can now be constructed via `Tridiag(3, 1, 2, 3)` again. Also, pattern matching works as before. However, due to space limitations, the implementation of `unapply` is not shown. The same abstractions are also provided for zero and identity matrices.

Translation

Next, the `translation` function in trait `SPL2CIR` is adapted. For this, we replace the case distinction for tridiagonal matrices with one that covers *all* Toeplitz matrices in general:

```
case Const(s @ Toeplitz1(n, coeffs)) =>
  def input(x: Int): TRep[Real] =
    if (x < 0 || x >= n) Real.zero else in(x)
  for (i <- 0 until n) {
    var res: TRep[Real] = 0
    for (j <- 0 until coeffs.length)
      res += coeffs(j) * input(i + j - s.center)
    out(i) = res }
```

The input array `in` contains the variables on which the stencil operates. The grid itself is modeled by a local function `input` which also implements the Dirichlet boundary condition. Next, we use a for-loop to successively apply the stencil to each lattice point. The result of such an application is given by the variable `res`. Its value is determined by iterating over the list of coefficients, thereby multiplying

each coefficient with the appropriate variable and adding the outcome to `res`. Attentive readers will have noticed that `res` is a *mutable* variable because it is declared using the `var` keyword. However, the generated code spares reassignments: in the IR graph, LMS makes effect dependencies between nodes explicit by adding an invisible state parameter to them; this is similar in spirit to single static-assignment (SSA) form and effectively turns the given CIR program into a functional one [7, p. 75].

Optimizations

Finally, we need to fix the optimizations in trait `SPLOpsExpOpt`. For example, the sum of two Toeplitz matrices is given by the sum of their coefficient lists:

```
override protected def spl_plus(lhs: Exp[SPL], rhs: Exp[SPL]) =
  (lhs, rhs) match {
    case (Const(Toeplitz1(n, xs)), Const(Toeplitz1(m, ys))) if n == m =>
      Toeplitz1(n, xs zip ys map { case (x, y) => x + y })
    ... }
```

Moreover, the scalar product can now be implemented as point-wise multiplication of the given scalar with the coefficient list:

```
override protected def spl_stimes(n: Real, spl: Exp[SPL]) =
  spl match {
    case Const(Toeplitz1(n, cs)) => Toeplitz1(m, cs.map(n * _))
    ... }
```

Since we introduced the standalone objects `I`, `Zero` and `Tridiag`, the patterns involving these extractors need not be changed. This concludes the implementation of stencil computations on a 1-dimensional grid.

3.7.4 Tensor Product of Toeplitz Matrices

Recurring Structures in Code

Section 3.7.1 briefly outlined the structure of a matrix $M \otimes M$ where M is tridiagonal. The corresponding CIR code contains a nested for-loop. The body of the innermost loop is of particular interest:

```
out(i * n + j) =
  aa * in((i - 1) * n + j - 1) +
  ab * in((i - 1) * n + j    ) +
  ac * in((i - 1) * n + j + 1) +
  ba * in( i      * n + j - 1) +
  bb * in( i      * n + j    ) +
  bc * in( i      * n + j + 1) +
  ca * in((i + 1) * n + j - 1) +
  cb * in((i + 1) * n + j    ) +
  cc * in((i + 1) * n + j + 1)
```

We find that there is a fixed access pattern to the input array `in`. A possible interpretation is that the outer loop variable `i` partitions `in` into disjoint chunks as it is multiplied with `n`, and the inner loop variable `j` iterates over the elements in each chunk. Additionally, the code hints at a relationship between the values the two loop counters attain and the coefficients found in $M \otimes M$. Naturally, each coefficient is a product of two factors. The first factor changes with `i` and the second one with `j`. All this suggests an implementation as 2-dimensional stencil computation. Therefore, we have to rearrange the variables given by the input vector in such a way that a 2-dimensional grid is obtained. Incidentally, the array access pattern conforms to the linearization of a matrix in row-major order.

Undoing Linearization

Let us generalize this example and consider the tensor product $M \otimes N$ of two Toeplitz matrices $M = M_{m\times m}$ and $N = N_{n\times n}$. The upper bound of the outer loop is given by m, which also happens to be the number of rows in aforementioned linearized matrix. In similar fashion, the inner loop's upper bound is determined by n, which gives the number of columns in the matrix. To exemplify, let $m = 3$ and $n = 2$. Then, the input vector has the dimension $m \cdot n = 6$. It represents the linearization of a 3×2 matrix:

$$\begin{bmatrix} x_0 & x_1 & x_2 & x_3 & x_4 & x_5 \end{bmatrix}^\top \quad \leftrightsquigarrow \quad \begin{bmatrix} x_0 & x_1 \\ x_2 & x_3 \\ x_4 & x_5 \end{bmatrix}.$$

That is, undoing the linearization of the input vector yields the rectangular grid on which the stencil operates.

Coefficient Matrices

Next, the challenge is to find a suitable *coefficient matrix*, which is the generalization of coefficient vectors. It turns out that this matrix can be conveniently constructed by multiplying c_M with the transposition of c_N. This yields an $m \times n$-matrix whose number of rows and columns is guaranteed to be odd. Hence, its center point is well-defined at row index $\lfloor m/2 \rfloor$ and column index $\lfloor n/2 \rfloor$. To illustrate, let us assume that

$$M := \begin{bmatrix} c & d & e \\ b & c & d \\ a & b & c \end{bmatrix} \quad \text{and} \quad N := \begin{bmatrix} \beta & \gamma \\ \alpha & \beta \end{bmatrix}.$$

Then, the respective coefficient vectors are

$$c_M = \begin{bmatrix} a & b & \boldsymbol{c} & d & e \end{bmatrix}^\top \quad \text{and} \quad c_N = \begin{bmatrix} \alpha & \boldsymbol{\beta} & \gamma \end{bmatrix}^\top.$$

The coefficient matrix for $M \otimes N$ is given by $c_M \cdot (c_N)^\top$. Its center point has the value $c\beta$. Just like before, the stencil applies the coefficient matrix to the

given 2-dimensional grid, while making sure to properly align the center point of the matrix with the currently regarded point in the grid. The coefficient matrix determines which neighboring points are taken into account and also tells their respective weighting.

Boundary Condition

Finally, we have to define a boundary condition. For our purpose, Dirichlet is once again the appropriate choice, that is, setting the values of all points outside the grid to 0.

Translation

Moving on to the implementation in Scala, the tensor product of two square Toeplitz matrices is represented by a new class `Toeplitz2` that extends `SPL`.

```
case class Toeplitz2(nL: Int, nR: Int, coeffs: List[List[Real]])
    extends SPL(nL * nR) {
  require(coeffs.length == 2 * nL - 1 &&
          coeffs.forall(_.length == 2 * nR - 1))
  val centerRows: Int = nL - 1
  val centerCols: Int = nR - 1 }
```

The properties `nL` and `nR` remember the dimension of the left-hand and right-hand arguments to the tensor. The coefficient matrix is represented as 2-dimensional list `coeffs`. Its row and column dimension are determined by `nL` and `nR`, and are tested for correctness by the `require` function. For convenience reasons, the index of `coeffs`' center point can be accessed via `centerRows` and `centerCols`.

Similar to `Toeplitz1`, we have to add a new case distinction to the pattern match within the `translate` function:

```
case Const(s @ Toeplitz2(n, m, css)) =>
  def input(row: Int)(col: Int): TRep[Real] =
    if (row < 0 || row >= n || col < 0 || col >= m) Real.zero
    else in(row * n + col)
  for (row <- 0 until n) {
    for (col <- 0 until m) {
      var res: TRep[Real] = 0
      for (i <- 0 until css.length) {
        for (j <- 0 until css(i).length) {
          val r = row + i - s.centerRows
          val c = col + j - s.centerCols
          res += css(i)(j) * input(r)(c) } }
      out(row * n + col) = res } }
```

The auxiliary function `input` is used to access the variables on the 2-dimensional rectangular grid, thereby assuming the Dirichlet boundary condition mentioned above. Next, we iterate over this grid using two nested for-loops. Within the body of the inner loop, we declare a mutable variable `res` that stores the result of applying the stencil to the current point on the grid. For this, we employ two

more for-loops to iterate over the coefficient matrix, multiplying a coefficient with the appropriate variable and adding the result to `res`. Finally, we update the output array `out` with the result `res` of performing one iteration in the stencil computation.

Noteworthy Benefits

This gives common code for all tensor products $M \otimes N$ of arbitrary square Toeplitz matrices M and N, which is noteworthy. What's more, it also shorter and less complicated than the specialized implementation for $\text{Tridiag}_n(a,b,c) \otimes \text{Tridiag}_n(a,b,c)$ presented in Section 3.7.1. Perhaps most importantly, the overhead of introducing stencil computations in the program generator does not pose a performance penalty on the generated code. All abstractions, such as the nested for-loops and the coefficient matrix, are translated away in the first program stage and will no longer be present in the second stage. Calls to the `input` function are replaced with array accesses and the index computations are inlined as constants.

Optimizations

The last task that remains to be done is to fix the optimizations in trait `SPL0psExp0pt`. Firstly, the tensor product of two instances of `Toeplitz1` yields an instance of `Toeplitz2`.

```
override protected def spl_tensor(lhs: Exp[SPL], rhs: Exp[SPL]) =
  (lhs, rhs) match {
    ...
    case (Const(Toeplitz1(n, xs)), Const(Toeplitz1(m, ys))) =>
      val css = xs.map(x => ys.map(y => x * y))
      Toeplitz2(n, m, css) }
```

Remember to place this pattern after more specific ones. The coefficient matrix `css` is obtained by multiplying the coefficient vector `xs` with the transposition of `ys`. Next, we complete the implementation of `spl_plus`:

```
protected def spl_plus(lhs: Exp[SPL], rhs: Exp[SPL]) =
  (lhs, rhs) match {
    case (Const(Toeplitz2(n1, m1, xss)), Const(Toeplitz2(n2, m2, yss)))
        if n1 == n2 && m1 == m2 =>
      val css = (xss zip yss) map { case (xs, ys) =>
        (xs zip ys) map { case (x, y) => x + y } }
      Toeplitz2(n1, m2, css)
    ... }
```

In order to do so, the coefficient matrices need to have the same number of rows and columns, which is tested by the guard. Taking the sum of two instances of `Toeplitz2` means taking the sum of the respective coefficient matrices; `css` is the result of adding `xss` to `yss` point-wise. This mechanism effectively implements constant folding and also causes that the expensive helper arrays mentioned in Section 3.6.2 are no longer needed here. Last but not least, we turn our attention to scalar multiplication `spl_stimes`:

```
override protected def spl_stimes(n: Real, spl: Exp[SPL]) =
  spl match {
    case Const(Toeplitz2(m1, m2, css)) =>
      Toeplitz2(m1, m2, css.map(_.map(n * _)))
    ... }
```

The implementation is straight-forward; multiplying a scalar and an instance of `Toeplitz2` is encoded by multiplying its coefficient matrix `css` with that scalar.

3.8 Unparsing to C Code

As mentioned in Section 2.2.2, the process of generating source code is decoupled from the implementation of any particular DSL. Instead, code generation needs to be triggered explicitly. This way, it is possible to unparse a given DSL program to multiple target languages. When doing so, the common workflow is roughly as follows: (1) traversing the IR graph of a given DSL program, (2) inspecting each encountered IR node, and (3) properly unparsing it to code fragments of the target language. LMS simplifies this task by providing a general code generation infrastructure. For instance, it already includes a scheduler and basic set of unparsers, in particular for Scala and C-like languages. The scheduler uses data and control dependencies encoded by the IR nodes to determine the order in which statements must be output, thus coming up with a correct program order [10, p. 97]. The only work for is to specialize existing unparsers for our own domain-specific types and needs. This is realized in trait `CGen`:

```
trait CGen
    extends CGenBase // Basic code generation infrastructure for C
    with CGenPrimitiveOps with CGenNumericOps // Numeric computations
    with CGenVariables with CGenArrayOps {    // Variables and Arrays
  val IR: CIRExp
  import IR._
```

Using modular mix-in composition, `CGen` is built from several existing unparsers. We also demand users of `CGen` to inject an implementation `IR` of CIR, since we need to examine the structure of CIR programs.

Atomic Expressions

Spire's `Real` data type sees frequent use in CIR code but there is no equivalent data representation to be found in C. The next best offering is `double`. Thus, it is imperative to convert a given `Real` value into the closest `double` value before unparsing. For this, we override the function `quote`, which comes from trait `CGen`. It takes an IR node (note the path-dependent type in the signature below) and returns an appropriate string representation of that expression, i.e. it yields the source code to be generated from the node:

```
override def quote(x: IR.Exp[Any]): String = x match {
  case Const(r: Real) => r.doubleValue.toString
  case _ => super.quote(x) }
```

This means that $\sqrt{2}$ is still approximated:

```
val sqrt2 = Real(2).sqrt
quote(Const(sqrt2)) == "1.4142135623730951"
```

But $(\sqrt{2})^2$ is replaced with an exact result:

```
val two = sqrt2 * sqrt2
quote(Const(two)) == "2.0"
```

Similarly, we also have define a suitable substitute for the `Real` data type itself. In other words, we have to remap `Real` to `double`:

```
override def remap[A](m: IR.Typ[A]): String = m.toString match {
  case "spire.math.Real" => "double"
  case _ => super.remap(m) }
```

Function `remap` is also inherited from `CGen`. Its input argument `m` is of type `IR.Typ[A]`, which serves as a runtime descriptor for the type `A`. We can now successfully unparse atomic expressions.

String Interpolation

In spirit of the string interpolation operator `s` introduced in Section 3.6.2, we may employ the custom "source code" interpolator `src` provided by LMS. It can be used in the same way as `s` but it quotes embedded expressions of type `IR.Exp[_]` and remaps ones of type `IR.Typ[_]` in lieu of simply inserting their canonical string representation via `toString`.

Composite Expressions

Atomic expressions rarely appear isolated in code. Rather, they are usually combined to form more complicated statements, such as value definitions or declarations. In LMS, every definition has an associated symbol. Function `emitNode` (inherited i.a. from `GCenBase`) produces target code for composite expressions. It requires a symbol `sym` and the corresponding definition `rhs`. Its return type is `Unit` as it expects us to write the unparsed code to a text-output stream named `stream`.

```
override def emitNode(sym: Sym[Any], rhs: Def[Any]): Unit = ...
```

We inspect `rhs` in a pattern match and unparse accordingly. For instance:

```
case Comment(s) => stream.println(src"/* $s */")
```

If the right-hand side `rhs` represents a comment, we discard its symbol, enclose the wrapped string `s` into the delimiters `/*` and `*/`, and print the result. That is, the CIR expression `comment("test")` will be converted into the C fragment `/* test */`.

Next, trait `CGenArrayOps` assumes arrays to be wrapped inside a `struct` that adheres to a certain interface. This does not apply in our case as we are going to use plain arrays and pointer dereferencing/arithmetic instead. For this reason, we have to customize the output for the IR nodes `ArrayApply`, `ArrayUpdate`, `ArraySlice` and `ArrayNew`. We exemplify the latter:

```
case a @ ArrayNew(n) => stream.println(src"${a.m} $sym[$n];")
```

`ArrayNew` encodes the creation of a new array via `NewArray`. Its element type is accessible via the identifier `m` and its length is given by the staged integer `n`. Internally, the expression `src"${a.m} $sym[$n];"` rewrites to:

```
remap(a.m) + " " + quote(sym) + "[" + quote(n) + "];"
```

Quoting a symbol results in a unique identifier that is constructed from the prefix "x" and the symbol's `id`; that is "x0", "x1", "x2" etc. Thus, the CIR expression

```
val x = NewArray[Real](unit(42))
```

is translated into the declaration

```
double x0[42];
```

whereby the identifier name x0 may vary.

Finally, all other IR nodes are handled by super traits:

```
case _ => super.emitNode(sym, rhs)
```

Code generation for the entire DSL program is performed by invoking the function `emitSource`, which is provided by trait `CGenBase`. It traverses the IR graph in dependency order and calls `emitNode` for every encountered statement.

Chapter 4

Use Cases

4.1 Setup

Guided by the principle of separating DSL interface and implementation, we place SPL programs in their own trait SPLApp. It only mixes in the interface SPLOps so that SPL programs are unable to reason about their own structure. Additionally, trait Breakdowns is also mixed in to make the breakdown function accessible.

```
trait SPLApp extends SPLOps with Breakdowns {
  def code: Rep[SPL] = /* SPL code */ }
```

Inside SPLApp, authors of an SPL program have to implement an abstract method named code with result type Rep[SPL].

To turn the code generator into an executable Scala program, we have to define a main entry point for the program. Therefore, we create a standalone object Main in which we put a method main with a proper signature:

```
object Main {
  def main(args: Array[String]): Unit = {
    val spl = new SPLApp with SPLOpsExpOpt
    val cir = new CIRExpOpt { }
    val translator = new SPL2CIR {
      override val source = spl
      override val target = cir }
    val fun = translator.translate(spl.code)
    ... } }
```

Inside main's body, the SPL application SPLApp is instantiated as spl, thereby mixing in the optimized implementation SPLOpsExpOpt, which is crucial. Afterwards, the optimized implementation for CIR is instantiated as cir. To set up the translation from SPL to CIR, we have to create a specialization of SPL2CIR and define source and target languages. Next, we can pass the SPL program, which is available as spl.code, to the translate function of trait SPL2CIR. The result is a CIR function fun, which can then be unparsed as described in Section 3.8.

4.2 Examples and Code Analysis

Identity Matrix

Let us begin with a simple SPL program for the 3×3 identity matrix. The appropriate implementation for method code in trait SPLApp is as follows:

```
def code: Rep[SPL] = I(3)
```

SPL programs are written in point-free style, that is, input and output vectors are silently omitted. The implementation of CIR will create two fresh new symbols x0 and x1 that represent the input and output array, respectively. This simple program is expanded into the following C code:

```
void i3(double* x0, double* x1) {
    double x3 = *(x0);
    *(x1) = x3;
    double x4 = *(x0 + 1);
    *(x1 + 1) = x4;
    double x5 = *(x0 + 2);
    *(x1 + 2) = x5; }
```

Note the similarities to the pseudo-code given in Section 3.6.1. The code loads the first component of the input vector via `*(x0)` and stores it in a variable x3, which is then written into the first element of the output array x1. The same strategy is applied to the remaining components. Arrays are accessed via pointers. The output code complies with SSA form, which effectively copies array elements into temporary variables to make them available for potential reuse. It also removes false dependencies, thus enabling the compiler to perform better register allocation and instruction scheduling [3, p. 132]. Reassignments in CIR code are turned into definitions of new variables in C code. The implementation also commits to three-address code (TAC), i.e. the right-hand side of an assignment can only be a unary or binary operation, or in other words, assignments represent the linearization of a syntax tree.

Optimizations

All optimizations defined in SPLOpsExpOpt and CIRExpOpt are enabled. For example, one may write an equivalent, but more convoluted program for I_3:

```
def code: Rep[SPL] = { Zero(3); I(1 + 2) * I(3) + Tridiag(3, 0, 0, 0) }
```

Even though this code is deliberately suboptimal, the program generator still outputs the same C program. First of all, the SPL program is sequentially composed of two matrix expressions. As `Zero(3)` does not contribute to the final result of the program, it is translated away by DCE. Furthermore, the multiplication with `I(3)` is left out, as well as the addition with the tridiagonal matrix because the generator is able to reason that $\text{Tridiag}_3(0,0,0) = 0_3$. Also, the computation

`1 + 2` is evaluated to `3` during generator time. In the end, the only expression that remains in the simplified SPL program is `I(3)`.

Handling Invalid Expressions

Another important aspect is that the generator rejects illegal expressions such as `I(3) + I(42)` or `Zero(0)`. This is due to matrices and operators in SPL investigating as to whether the given arguments are valid. If this requirement fails, an exception is thrown, which causes the program generator to terminate so that no C code is produced.

Tridiagonal Matrix

Let us continue with a more advanced example, namely the computational kernel for a tridiagonal matrix $\text{Tridiag}_3(\sqrt{2}/2, \sqrt{2}, \sqrt{2}/2)$:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} := \begin{bmatrix} \sqrt{2} & \sqrt{2}/2 & 0 \\ \sqrt{2}/2 & \sqrt{2} & \sqrt{2}/2 \\ 0 & \sqrt{2}/2 & \sqrt{2} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

A domain expert may write:

```
def code: Rep[SPL] = {
  val sqrt2 = Real(2).sqrt
  Tridiag(3, sqrt2 / 2, sqrt2, sqrt2 / 2) }
```

This yields the following C code:

```
void tridiag3(double* x0, double* x1) {
    /* y_0 = √2 · x_0 + √2/2 · x_1 */
    double x3 = *(x0);
    double x4 = 1.4142135623730951 * x3;
    double x5 = *(x0 + 1);
    double x6 = 0.7071067811865476 * x5;
    double x7 = x4 + x6;
    *(x1) = x7;
    /* y_1 = √2/2 · x_0 + √2 · x_1 + √2/2 · x_2 */
    double x10 = 0.7071067811865476 * x3;
    double x11 = 1.4142135623730951 * x5;
    double x12 = x10 + x11;
    double x8 = *(x0 + 2);
    double x13 = 0.7071067811865476 * x8;
    double x14 = x12 + x13;
    *(x1 + 1) = x14;
    /* y_2 = √2/2 · x_1 + √2 · x_2 */
    double x16 = 1.4142135623730951 * x8;
    double x17 = x6 + x16;
    *(x1 + 2) = x17; }
```

We notice that the overhead of extracting `Real(2).sqrt` into its own variable `sqrt2` is gone. Also, static computations like `sqrt2 / 2` have been replaced with their evaluation result and multiplications with 0 have been eliminated. The encoding of real numbers as values of type `Real` is no longer present, either. The C program uses an (approximated) `double` representation instead. Furthermore, the for-loop used internally by the implementation of CIR has been unrolled. And although the code for both y_1 and y_2 requires the result of $\sqrt{2}/2 \cdot x_2$, we do not perform this computation twice. Instead, CSE kicks in and simply reuses the variable x6 in the computation for y_2.

Sum of Tensor Products

Due to the use of stencil computations and constant folding in the generator, complex expressions such as the sum of tensor products of Toeplitz matrices go without any intermediate storage. We use

$$\text{Tridiag}_3(1,-2,1) \otimes \text{I}_3 + \text{I}_3 \otimes \text{Tridiag}_3(1,-2,1)$$

as an example. The result is a 9×9 matrix

$$\begin{bmatrix} A & \text{I}_3 & 0_3 \\ \text{I}_3 & A & \text{I}_3 \\ 0_3 & \text{I}_3 & A \end{bmatrix} \quad \text{where} \quad A := \text{Tridiag}_3(1,-4,1) = \begin{bmatrix} -4 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & -4 \end{bmatrix}.$$

A matching SPL implementation is given by

```
def code: Rep[SPL] = {
  val (t, i) = (Tridiag(3, 1, -2, 1), I(3))
  (t tensor i) + (i tensor t) }
```

This yields the following C code:

```
void f(double* x0, double* x1) {
    double x3 = *(x0);
    double x4 = -4.0 * x3;
    double x5 = *(x0 + 1);
    double x6 = x4 + x5;
    double x8 = *(x0 + 3);
    double x9 = x6 + x8;
    *(x1) = x9;
    double x10 = *(x0 + 4);
    double x7 = *(x0 + 2);
    double x16 = -4.0 * x5;
    double x17 = x3 + x16;
    double x18 = x17 + x7;
    double x19 = x18 + x10;
    *(x1 + 1) = x19;
    double x11 = *(x0 + 5);
    double x21 = -4.0 * x7;
    double x22 = x5 + x21;
    double x23 = x22 + x11;
    *(x1 + 2) = x23;
    double x12 = *(x0 + 6);
    double x25 = -4.0 * x8;
    double x26 = x3 + x25;
    double x27 = x26 + x10;
    double x28 = x27 + x12;
    *(x1 + 3) = x28;
    double x13 = *(x0 + 7);
    double x30 = x5 + x8;
    double x31 = -4.0 * x10;
    double x32 = x30 + x31;
```

```
    double x33 = x32 + x11;
    double x34 = x33 + x13;
    *(x1 + 4) = x34;
    double x14 = *(x0 + 8);
    double x36 = x7 + x10;
    double x37 = -4.0 * x11;
    double x38 = x36 + x37;
    double x39 = x38 + x14;
    *(x1 + 5) = x39;
    double x41 = -4.0 * x12;
    double x42 = x8 + x41;
    double x43 = x42 + x13;
    *(x1 + 6) = x43;
    double x45 = x10 + x12;
    double x46 = -4.0 * x13;
    double x47 = x45 + x46;
    double x48 = x47 + x14;
    *(x1 + 7) = x48;
    double x50 = x11 + x13;
    double x51 = -4.0 * x14;
    double x52 = x50 + x51;
    *(x1 + 8) = x52; }
```

Just as claimed, no temporary array storage is used, which is important for the efficiency of the code generated from breakdown rules like $\text{ResidueLaplace}_{n,\omega}$. Apart from that, the presented C program has the same structure and properties as discussed before.

Interim Conclusion

The small numerical kernels considered so far are fairly high optimized, which already makes them useful for practical applications [1, p. 8]: using a divide-and-conquer approach, a recursive algorithm could break down a larger problem into smaller ones; at some point, the recursion is terminated and aforementioned small kernels are employed to implement base cases for both divide and conquer phases; these kernels then usually make up most of the runtime as they contain the majority of floating-point operations.

Matrix Multiplication

Contrary to the sum of tensor products, the next use case requires intermediate storage. We compute the product of two small tridiagonal matrices $\text{Tridiag}_2(1,2,3)$ and $\text{Tridiag}_2(4,5,6)$:

```
def code: Rep[SPL] = Tridiag(2, 1, 2, 3) * Tridiag(2, 4, 5, 6)
```

This translates into the following C code:

```
void f(double* x0, double* x1) {
    /* Temporary storage */
    double x3[2];
    /* x3 = Tridiag_2(4,5,6)·x0 */
    double x5 = *(x0);
    double x6 = 5.0 * x5;
    double x7 = *(x0 + 1);
    double x8 = 6.0 * x7;
    double x9 = x6 + x8;
    *(x3) = x9;
    double x11 = 4.0 * x5;
    double x12 = 5.0 * x7;
    double x13 = x11 + x12;
    *(x3 + 1) = x13;
    /* x1 = Tridiag_2(1,2,3)·x3 */
    double x17 = *(x3);
    double x18 = 2.0 * x17;
    double x19 = 3.0 * x13;
    double x20 = x18 + x19;
    *(x1) = x20;
    double x22 = 2.0 * x13;
    double x23 = x17 + x22;
    *(x1 + 1) = x23; }
```

Notice the declaration of the local array `x3`. It serves as the output array for $\text{Tridiag}_2(4,5,6)$ and as input for $\text{Tridiag}_2(1,2,3)$. Apparently, constant folding of the tridiagonal matrices cannot be applied here, at least not with the current set of stencil computations implemented, which means the creation of `x3` is inevitable.

Array Scalarization

One could still turn the array into scalar variables, though. However, this optimization is currently not supported by the generator. The root cause is that the `translation` function hard codes the use of dynamic arrays of type `TRep[Array[Real]]` in its signature. But array scalarization would require arrays of dynamic variables, i.e. `Array[TRep[Real]]`. Hard coding the latter is no solution, either, as this would result in the generation of C functions with unwieldy signatures: there would be one formal parameter of type `double*` for every element in the input/output arrays. Addressing this issue includes restructuring the implementation of the generator, in particular the introduction of a special array data type that abstracts over the choice of staging decisions (c.f. the remarks in Section 3.6.3). This way, one could provide runtime parameters to decide whether to scalarize arrays in the generated code. The invocation of `translate` in Section 4.1, which determines the signature of the generated C function, could be advised to stage the arrays, while the recursive calls to `translate` in Section 3.6 could be parameterized to scalarize them.

Multigrid Cycle

Finally, we present an SPL program that yields code for a multigrid cycle $\text{MGCycle}_{3,0.125,1}$:

```
def code: Rep[SPL] = breakdown(MGCycle(3, 0.125, 1))
```

We only need to give an SPL specification `MGCycle(3, 0.125, 1)` of the multigrid cycle. Using the `breakdown` function as demonstrated above causes the specification to be fully expanded until it consists solely of terminals. The resulting C code is listed in Appendix A on Page 73. In comparison to SPIRAL [1, p. 9], our generated code is much more extensive. The reason for this is twofold. Firstly, SPIRAL does not convert to TAC, i.e. the right-hand side of assignments can consist of large expressions [1, p. 8]. But the more pivotal cause leads back to the lack of further sophisticated optimizations in our implementation. Most notably, many operations, such as matrix multiplication, addition or horizontal stacking, often entail the generation of helper arrays to store intermediate computation results. The lack of array scalarization as discussed above further contributes to this issue.

Chapter 5

Conclusions

Summary

In this thesis we have demonstrated that it is possible to build a fully-fledged—albeit very simple and highly specialized—code generator for a multigrid solver using a DSL-centric approach, as pursued by SPIRAL and the ExaStencils project.

At the core of the generator sits a two-layered architecture of internal DSLs to represent algorithmic and domain-specific knowledge on different levels of abstraction. Coupled with a rule-based rewriting system of DSL expressions, this enables the synthesis of suitable solving algorithms from an abstract mathematical problem specification as starting point. The application of domain-specific and classic compiler optimizations is performed gradually on both DSL layers with an intermediate translation step between those layers. Among the supported optimizations are CSE, DCE, constant folding, algebraic simplification, copy propagation and precomputation.

The generator outputs fully unrolled standard C code that complies with SSA and TAC form. We have shown that the generator is capable of producing code for an entire multigrid cycle and also for simpler but in return more efficient numerical kernels. As argued in Section 4.2, the latter could be useful for applications in other program synthesis systems or high-performance libraries.

Future Work

Although we have seen encouraging results, the produced code for a multigrid cycle is much more extensive than the SPIRAL-generated counterpart. The main reason is that many frequently occurring operations, such as matrix multiplications, require helper arrays to store intermediate results. Thus, it is desirable to fuse such operations into a single matrix already during generator time. In the case of sums of tensor products of Toeplitz matrices, this goal has been reached by leveraging the structural regularity found in those matrices and the use of stencil computations. This leads us to believe that similar results could be achieved for other operations as well, likely with different kinds of stencils that are yet to be investigated.

Another shortcoming is that we do not abstract over the choice of staging decisions to support different code styles and data representations. Concretely,

this would allow for scalarization of arrays based on runtime parameters, which has the potential to enable more advanced and currently unfeasible optimizations. Deriving such abstractions and accompanying meta programming techniques has been an ongoing, independent research effort [4] at the time of this writing. Incorporating these techniques into our generator could therefore prove beneficial.

Finally, the current set of rewrite rules does not have any degrees of freedom involved. For this reason, implementing a simple top-down parser of DSL expressions sufficed. However, as there are plans to support more complex multigrid methods by developing a parameterized meta rule system [1, p. 10], we will have to extend our implementation in such a way that it supports search over algorithmic choices, e.g. via auto tuning or machine learning.

Appendix A

Generated Code for a Multigrid Cycle

C code output by the program generator for the SPL specification $\mathrm{MGCycle}_{3,0.125,1}$:

```
void mgcycle(double* x0, double* x1) {
    double x3[18];
    double* x5 = (x0 + 9);
    double x6[18];
    double x7[18];
    double x10 = *(x0 + 9);
    double x11 = 0.125 * x10;
    *(x7) = x11;
    double x12 = *(x0 + 10);
    double x21 = 0.125 * x12;
    *(x7 + 1) = x21;
    double x13 = *(x0 + 11);
    double x23 = 0.125 * x13;
    *(x7 + 2) = x23;
    double x14 = *(x0 + 12);
    double x25 = 0.125 * x14;
    *(x7 + 3) = x25;
    double x15 = *(x0 + 13);
    double x27 = 0.125 * x15;
    *(x7 + 4) = x27;
    double x16 = *(x0 + 14);
    double x29 = 0.125 * x16;
    *(x7 + 5) = x29;
    double x17 = *(x0 + 15);
    double x31 = 0.125 * x17;
    *(x7 + 6) = x31;
    double x18 = *(x0 + 16);
    double x33 = 0.125 * x18;
    *(x7 + 7) = x33;
    double x19 = *(x0 + 17);
    double x35 = 0.125 * x19;
    *(x7 + 8) = x35;
    double* x38 = (x7 + 9);
    *(x38) = x10;
    *(x38 + 1) = x12;
    *(x38 + 2) = x13;
    *(x38 + 3) = x14;
    *(x38 + 4) = x15;
    *(x38 + 5) = x16;
    *(x38 + 6) = x17;
    *(x38 + 7) = x18;
    *(x38 + 8) = x19;
    double x53 = *(x0);
    double x54 = 0.5 * x53;
    double x55 = *(x0 + 1);
    double x56 = 0.125 * x55;
    double x57 = x54 + x56;
    double x59 = *(x0 + 3);
    double x60 = 0.125 * x59;
    double x61 = x57 + x60;
    *(x6) = x61;
    double x68 = 0.125 * x53;
    double x69 = 0.5 * x55;
    double x70 = x68 + x69;
    double x58 = *(x0 + 2);
    double x71 = 0.125 * x58;
```

```
56      double x72 = x70 + x71;
        double x62 = *(x0 + 4);
        double x73 = 0.125 * x62;
        double x74 = x72 + x73;
60      *(x6 + 1) = x74;
        double x76 = 0.5 * x58;
        double x77 = x56 + x76;
        double x63 = *(x0 + 5);
        double x78 = 0.125 * x63;
65      double x79 = x77 + x78;
        *(x6 + 2) = x79;
        double x81 = 0.5 * x59;
        double x82 = x68 + x81;
        double x83 = x82 + x73;
70      double x64 = *(x0 + 6);
        double x84 = 0.125 * x64;
        double x85 = x83 + x84;
        *(x6 + 3) = x85;
        double x87 = x56 + x60;
75      double x88 = 0.5 * x62;
        double x89 = x87 + x88;
        double x90 = x89 + x78;
        double x65 = *(x0 + 7);
        double x91 = 0.125 * x65;
80      double x92 = x90 + x91;
        *(x6 + 4) = x92;
        double x94 = x71 + x73;
        double x95 = 0.5 * x63;
        double x96 = x94 + x95;
85      double x66 = *(x0 + 8);
        double x97 = 0.125 * x66;
        double x98 = x96 + x97;
        *(x6 + 5) = x98;
        double x100 = 0.5 * x64;
90      double x101 = x60 + x100;
        double x102 = x101 + x91;
        *(x6 + 6) = x102;
        double x104 = x73 + x84;
        double x105 = 0.5 * x65;
95      double x106 = x104 + x105;
        double x107 = x106 + x97;
        *(x6 + 7) = x107;
        double x109 = x78 + x91;
        double x110 = 0.5 * x66;
100     double x111 = x109 + x110;
        *(x6 + 8) = x111;
        double* x114 = (x6 + 9);
        *(x114) = 0.0;
        *(x114 + 1) = 0.0;
105     *(x114 + 2) = 0.0;
        *(x114 + 3) = 0.0;
        *(x114 + 4) = 0.0;
        *(x114 + 5) = 0.0;
        *(x114 + 6) = 0.0;
110     *(x114 + 7) = 0.0;
        *(x114 + 8) = 0.0;
        double x127 = *(x6);
        double x128 = *(x7);
        double x129 = x127 + x128;
115     *(x3) = x129;
        double x131 = *(x6 + 1);
        double x132 = *(x7 + 1);
        double x133 = x131 + x132;
        *(x3 + 1) = x133;
120     double x135 = *(x6 + 2);
        double x136 = *(x7 + 2);
        double x137 = x135 + x136;
        *(x3 + 2) = x137;
        double x139 = *(x6 + 3);
125     double x140 = *(x7 + 3);
        double x141 = x139 + x140;
        *(x3 + 3) = x141;
        double x143 = *(x6 + 4);
        double x144 = *(x7 + 4);
130     double x145 = x143 + x144;
        *(x3 + 4) = x145;
        double x147 = *(x6 + 5);
        double x148 = *(x7 + 5);
        double x149 = x147 + x148;
135     *(x3 + 5) = x149;
        double x151 = *(x6 + 6);
        double x152 = *(x7 + 6);
```

```
138     double x153 = x151 + x152;
        *(x3 + 6) = x153;
140     double x155 = *(x6 + 7);
        double x156 = *(x7 + 7);
        double x157 = x155 + x156;
        *(x3 + 7) = x157;
        double x159 = x111 + x35;
145     *(x3 + 8) = x159;
        double x161 = *(x6 + 9);
        double x162 = *(x7 + 9);
        double x163 = x161 + x162;
        *(x3 + 9) = x163;
150     double x165 = *(x6 + 10);
        double x166 = *(x7 + 10);
        double x167 = x165 + x166;
        *(x3 + 10) = x167;
        double x169 = *(x6 + 11);
155     double x170 = *(x7 + 11);
        double x171 = x169 + x170;
        *(x3 + 11) = x171;
        double x173 = *(x6 + 12);
        double x174 = *(x7 + 12);
160     double x175 = x173 + x174;
        *(x3 + 12) = x175;
        double x177 = *(x6 + 13);
        double x178 = *(x7 + 13);
        double x179 = x177 + x178;
165     *(x3 + 13) = x179;
        double x181 = *(x6 + 14);
        double x182 = *(x7 + 14);
        double x183 = x181 + x182;
        *(x3 + 14) = x183;
170     double x185 = *(x6 + 15);
        double x186 = *(x7 + 15);
        double x187 = x185 + x186;
        *(x3 + 15) = x187;
        double x189 = *(x6 + 16);
175     double x190 = *(x7 + 16);
        double x191 = x189 + x190;
        *(x3 + 16) = x191;
        double x193 = *(x6 + 17);
        double x194 = *(x7 + 17);
180     double x195 = x193 + x194;
        *(x3 + 17) = x195;
        double x200[9];
        double x202 = *(x3);
        double x203 = -4.0 * x202;
185     double x204 = *(x3 + 1);
        double x205 = x203 + x204;
        double x206 = *(x3 + 2);
        double x207 = *(x3 + 3);
        double x208 = x205 + x207;
190     double x209 = *(x3 + 4);
        double x210 = *(x3 + 5);
        double x211 = *(x3 + 6);
        double x212 = *(x3 + 7);
        double x213 = *(x3 + 8);
195     double x214 = *(x3 + 9);
        double x215 = x208 + x214;
        *(x200) = x215;
        double x217 = -4.0 * x204;
        double x218 = x202 + x217;
200     double x219 = x218 + x206;
        double x220 = x219 + x209;
        double x221 = *(x3 + 10);
        double x222 = x220 + x221;
        *(x200 + 1) = x222;
205     double x224 = -4.0 * x206;
        double x225 = x204 + x224;
        double x226 = x225 + x210;
        double x227 = *(x3 + 11);
        double x228 = x226 + x227;
210     *(x200 + 2) = x228;
        double x230 = -4.0 * x207;
        double x231 = x202 + x230;
        double x232 = x231 + x209;
        double x233 = x232 + x211;
215     double x234 = *(x3 + 12);
        double x235 = x233 + x234;
        *(x200 + 3) = x235;
        double x237 = x204 + x207;
        double x238 = -4.0 * x209;
```

```
double x239 = x237 + x238;
double x240 = x239 + x210;
double x241 = x240 + x212;
double x242 = *(x3 + 13);
double x243 = x241 + x242;
*(x200 + 4) = x243;
double x245 = x206 + x209;
double x246 = -4.0 * x210;
double x247 = x245 + x246;
double x248 = x247 + x213;
double x249 = *(x3 + 14);
double x250 = x248 + x249;
*(x200 + 5) = x250;
double x252 = -4.0 * x211;
double x253 = x207 + x252;
double x254 = x253 + x212;
double x255 = *(x3 + 15);
double x256 = x254 + x255;
*(x200 + 6) = x256;
double x258 = x209 + x211;
double x259 = -4.0 * x212;
double x260 = x258 + x259;
double x261 = x260 + x213;
double x262 = *(x3 + 16);
double x263 = x261 + x262;
*(x200 + 7) = x263;
double x265 = x210 + x212;
double x266 = -4.0 * x213;
double x267 = x265 + x266;
double x268 = x267 + x195;
*(x200 + 8) = x268;
double x272[1];
double x274 = *(x200 + 4);
*(x272) = x274;
double x278[1];
double x280 = 0.25 * x274;
*(x278) = x280;
double x284[9];
double x286 = *(x278 + 4);
*(x284) = x286;
double x290 = 2.0 * x286;
double x291 = *(x284 + 1);
double x292 = x290 + x291;
double x293 = *(x284 + 2);
double x294 = *(x284 + 3);
double x295 = x292 + x294;
double x296 = *(x284 + 4);
double x297 = 0.5 * x296;
double x298 = x295 + x297;
double x299 = *(x284 + 5);
double x300 = *(x284 + 6);
double x301 = *(x284 + 7);
double x302 = *(x284 + 8);
*(x1) = x298;
double x304 = 2.0 * x291;
double x305 = x286 + x304;
double x306 = x305 + x293;
double x307 = 0.5 * x294;
double x308 = x306 + x307;
double x309 = x308 + x296;
double x310 = 0.5 * x299;
double x311 = x309 + x310;
*(x1 + 1) = x311;
double x313 = 2.0 * x293;
double x314 = x291 + x313;
double x315 = x314 + x297;
double x316 = x315 + x299;
*(x1 + 2) = x316;
double x318 = 0.5 * x291;
double x319 = x286 + x318;
double x320 = 2.0 * x294;
double x321 = x319 + x320;
double x322 = x321 + x296;
double x323 = x322 + x300;
double x324 = 0.5 * x301;
double x325 = x323 + x324;
*(x1 + 3) = x325;
double x327 = 0.5 * x286;
double x328 = x327 + x291;
double x329 = 0.5 * x293;
double x330 = x328 + x329;
double x331 = x330 + x294;
```

```
302     double x332 = 2.0 * x296;
        double x333 = x331 + x332;
        double x334 = x333 + x299;
305     double x335 = 0.5 * x300;
        double x336 = x334 + x335;
        double x337 = x336 + x301;
        double x338 = 0.5 * x302;
        double x339 = x337 + x338;
310     *(x1 + 4) = x339;
        double x341 = x318 + x293;
        double x342 = x341 + x296;
        double x343 = 2.0 * x299;
        double x344 = x342 + x343;
315     double x345 = x344 + x324;
        double x346 = x345 + x302;
        *(x1 + 5) = x346;
        double x348 = x294 + x297;
        double x349 = 2.0 * x300;
320     double x350 = x348 + x349;
        double x351 = x350 + x301;
        *(x1 + 6) = x351;
        double x353 = x307 + x296;
        double x354 = x353 + x310;
325     double x355 = x354 + x300;
        double x356 = 2.0 * x301;
        double x357 = x355 + x356;
        double x358 = x357 + x302;
        *(x1 + 7) = x358;
330     double x360 = x297 + x299;
        double x361 = x360 + x301;
        double x362 = 2.0 * x302;
        double x363 = x361 + x362;
        *(x1 + 8) = x363;
335     double* x370 = (x3 + 9);
        double* x371 = (x1 + 9);
        *(x371) = x214;
        *(x371 + 1) = x221;
        *(x371 + 2) = x227;
340     *(x371 + 3) = x234;
        *(x371 + 4) = x242;
        *(x371 + 5) = x249;
        *(x371 + 6) = x255;
        *(x371 + 7) = x262;
345     *(x371 + 8) = x195; }
```

Bibliography

[1] Matthias Bolten, Franz Franchetti, Paul H. J. Kelly, Christian Lengauer, and Marcus Mohr. “Algebraic Description and Automatic Generation of Multigrid Methods in SPIRAL.”
In: *Concurrency and Computation: Practice and Experience* 29.17 (Sept. 2017): *Special Issue on Advanced Stencil-Code Engineering*, 4105:1–4105:11.
Preprint available at http://www.infosun.fim.uni-passau.de/publications/docs/BFKLM16ccpe.pdf.

[2] Stefan Kronawitter and Christian Lengauer.
Optimizations Applied by the ExaStencils Code Generator.
Technical Report MIP-1502. University of Passau, Jan. 2015.
URL: http://www.infosun.fim.uni-passau.de/publications/docs/KroLe2015tr.pdf.

[3] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. “Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries.” In: *International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM. 2013, pp. 125–134. ISBN: 978-1-4503-2373-4. DOI: 10.1145/2517208.2517228.
Preprint available at http://spiral.ece.cmu.edu:8080/pub-spiral/pubfile/paper_170.pdf.

[4] Georg Ofenbeck. “Generic Programming in Space and Time. Abstractions for High Performance Code Generation.”
PhD thesis. Eidgenössische Technische Hochschule Zürich, 2017.

[5] Christian Lengauer, Sven Apel, Matthias Bolten, Armin Größlinger, Frank Hannig, Harald Köstler, Ulrich Rüde, Jürgen Teich, Alexander Grebhahn, Stefan Kronawitter, Sebastian Kuckuk, Hannah Rittich, and Christian Schmitt.
“ExaStencils: Advanced Stencil-Code Engineering.” In:
Euro-Par 2014: Parallel Processing Workshops, Part II. Springer, 2014, pp. 553–564. ISBN: 978-3-319-14313-2.
DOI: 10.1007/978-3-319-14313-2_47.

[6] Markus Püschel, Franz Franchetti, and Yevgen Voronenko. “SPIRAL.” In: *Encyclopedia of Parallel Computing*. Springer, 2011. Chap. 709, pp. 1920–1933. ISBN: 978-0-387-09765-7. DOI: 10.1007/978-0-387-09766-4. Preprint available at http://spiral.ece.cmu.edu:8080/pub-spiral/pubfile/paper_146.pdf.

[7] Tiark Rompf. “Lightweight Modular Staging and Embedded Compilers: Abstraction without Regret for High-Level High-Performance Programming.” PhD thesis. École Polytechnique Fédérale de Lausanne, 2012. URL: https://infoscience.epfl.ch/record/180642/files/EPFL_TH5456.pdf.

[8] Tiark Rompf. *LMS: Program Generation and Embedded Compilers in Scala*. URL: https://scala-lms.github.io/ (visited on 09/11/2017).

[9] Tiark Rompf and Martin Odersky. “Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs.” In: *ACM Sigplan Notices*. Vol. 55. 6. ACM. 2012, pp. 121–130. DOI: 10.1145/2184319.2184345. Preprint of a previous version available at https://infoscience.epfl.ch/record/150347/files/gpce63-rompf.pdf.

[10] Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. “Building-Blocks for Performance Oriented DSLs.” In: *Proceedings IFIP Working Conference on Domain-Specific Languages*. Ed. by Olivier Danvy and Chung-chieh Shan. Vol. 66. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2011, pp. 93–117. DOI: 10.4204/EPTCS.66.5. URL: https://ppl.stanford.edu/papers/dsl11-rompf.pdf.

[11] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. 3rd ed. Artima Inc., 2016.

[12] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. “Scala-Virtualized: Linguistic Reuse for Deep Embeddings.” In: *Higher Order and Symbolic Computation* August-September (2013), pp. 1–43. ISSN: 1388-3690. DOI: 10.1007/s10990-013-9096-9. Preprint available at https://infoscience.epfl.ch/record/197945/files/hosc2013.pdf.

[13] Tiark Rompf. *Virtualized Scala Reference*.
URL: https://github.com/TiarkRompf/scala-virtualized/wiki/Virtualized-Scala-Reference (visited on 09/11/2017).

List of Acronyms

CIR C internal representation

CSE common subexpression elimination

DCE dead code elimination

DFT discrete Fourier transform

DSL domain-specific language

DSP digital signal processing

IR intermediate representation

JVM Java Virtual Machine

LMS Lightweight Modular Staging

MSP multi-stage programming

PDE partial differential equation

SPL Signal Processing Language

SSA single static-assignment

TAC three-address code

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich diese Bachelor-Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet habe, sowie dass ich die Bachelor-Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, den 22. September 2017

Sebastian Schweikl