Universidade Estadual de Campinas
Instituto de Computação

Luciano Gigantelli Zago

A pattern generation language for MLIR compiler matching and rewriting

Uma linguagem para geração de padrões para detecção e reescrita em compiladores MLIR

CAMPINAS
2024

Luciano Gigantelli Zago

A pattern generation language for MLIR compiler matching and rewriting

Uma linguagem para geração de padrões para detecção e reescrita em compiladores MLIR

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Guido Costa Souza de Araújo
Co-supervisor/Coorientador: Dr. Márcio Machado Pereira

Este exemplar corresponde à versão final da Dissertação defendida por Luciano Gigantelli Zago e orientada pelo Prof. Dr. Guido Costa Souza de Araújo.

CAMPINAS
2024

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

Z13p

Zago, Luciano Gigantelli, 1997-
A pattern generation language for MLIR compiler matching and rewriting / Luciano Gigantelli Zago. – Campinas, SP : [s.n.], 2024.

Orientador: Guido Costa Souza de Araújo.
Coorientador: Márcio Machado Pereira.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Linguagem de programação (Computadores). 2. Compiladores (Computadores). 3. Geradores de código. I. Araújo, Guido Costa Souza de, 1962-. II. Pereira, Márcio Machado, 1959-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações Complementares

Título em outro idioma: Uma linguagem para geração de padrões para detecção e reescrita em compiladores MLIR
Palavras-chave em inglês:
Programming languages (Electronic computers)
Compilers (Electronic computers)
Code generators
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Guido Costa Souza de Araújo [Orientador]
Wesley Attrot
Alexandro José Baldassin
Data de defesa: 10-01-2024
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0003-4417-163X
- Currículo Lattes do autor: http://lattes.cnpq.br/7367165646860669

Universidade Estadual de Campinas
Instituto de Computação

Luciano Gigantelli Zago

A pattern generation language for MLIR compiler matching and rewriting

Uma linguagem para geração de padrões para detecção e reescrita em compiladores MLIR

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo
  IC/UNICAMP
- Prof. Dr. Wesley Attrot
  CCE/UEL
- Prof. Dr. Alexandro José Baldassin
  IGCE/UNESP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 10 de janeiro de 2024

Acknowledgements

I extend my sincere gratitude to Dr. Guido Araújo, my esteemed advisor, whose guidance and support have been pivotal in shaping this master's thesis. His insights and unwavering commitment to academic excellence have profoundly influenced the trajectory of my research, for which I am truly thankful.

I would like to express my appreciation to my co-advisor, Dr. Márcio Pereira, whose invaluable contributions and constructive feedback have significantly enhanced the depth and quality of this work. His thorough and thoughtful reviews have been instrumental in refining the quality and rigor of this dissertation.

I also want to thank Dr. Hervé Yviquel for his insights and substantive contributions to the project. His expertise has really influenced the direction of my research.

Special acknowledgment is extended to my project colleague and friend, Vinicius Espindola. Our collaborative efforts have not only rendered a stimulating research but have also fostered an enriching academic environment.

Heartfelt gratitude is due to my parents for their constant support, unwavering encouragement, and enduring belief in my academic pursuits. Their sacrifices and parental guidance have been a constant source of motivation, shaping my academic trajectory.

Finally, I wish to express my profound appreciation to my girlfriend for her understanding, patience, and constant support. Her encouragement and belief in my capabilities have provided immeasurable strength, contributing significantly to the successful completion of this academic venture.

This study was financed by The Brazilian National Council for Scientific and Technological Development (CNPq), grant #131197/2021-5.

Resumo

O casamento e reescrita de padrões é uma etapa de otimização do compilador que identifica idiomas de código predefinidos e os substitui por código otimizado, resultando em ganhos de desempenho em várias aplicações. Avanços recentes levaram a ferramentas que agilizam o casamento de padrões e otimizações de reescrita. Uma dessas técnicas, *Source Matching and Rewriting (SMR)*, emprega uma abordagem centrada no usuário e baseada em código-fonte, eliminando a necessidade de casamento de padrões internamente dentro do compilador. No entanto, alcançar uma cobertura abrangente de casamento de padrões com SMR requer uma especificação meticulosa de todas as variações linguísticas possíveis dos padrões por parte do usuário, uma tarefa trabalhosa e propensa a erros.

Esta pesquisa apresenta a *Pattern Generation Language (PGL)*, uma linguagem que visa simplificar a geração automática de variações de padrões. PGL é uma linguagem de alto nível que permite ao usuário definir padrões de programa que podem ser casados e reescritos por meio do SMR. Adicionalmente, desenvolvemos o *PGL Compiler (PGC)*, uma ferramenta compatível com SMR que automatiza a criação de variações idiomáticas e a síntese de padrões definidos na linguagem PGL. Embora o PGC se concentre principalmente na geração de padrões de entrada para SMR, sua flexibilidade permite a adaptação para outras ferramentas de casamento e reescrita de padrões, mostrando sua versatilidade e potencial para diversas aplicações.

Os resultados experimentais evidenciam que PGL é capaz de identificar padrões em códigos escritos em Fortran e C, substituindo-os por chamadas à biblioteca BLAS e, assim, aprimorando o desempenho dos programas.

Abstract

Pattern Matching and Rewriting is a compiler optimization step that identifies predefined code idioms and replaces them with optimized code, offering performance gains across various applications. Recent research advances have led to tools that expedite pattern matching and rewriting optimizations. One such technique, Source Matching and Rewriting (SMR), employs a user-centric, source-code-based approach to pattern matching and rewriting, eliminating the need for specialized compiler intervention. However, achieving comprehensive pattern-matching coverage with SMR requires meticulous specification of all possible language variations by the user, a laborious and error-prone task.

This research introduces the Pattern Generation Language (PGL), which is aimed at simplifying the automatic generation of pattern variations. PGL is a high-level language that enables the user to define program patterns that can be matched and rewritten by SMR. Additionally, this work developed the PGL Compiler (PGC), an SMR-compatible tool that automates the creation of idiomatic variations and the synthesis of patterns defined in the PGL language. While PGC primarily focuses on generating input patterns for SMR, its flexibility allows adaptation for other pattern-matching and rewriting tools, showcasing its versatility and potential for diverse applications.

The experimental results demonstrate that PGL can identify patterns in Fortran and C code and replace them with calls to the BLAS library to enhance program performance.

List of Figures

List of Tables

List of Abbreviations and Acronyms

2MM	Two matrix-matrix multiplications in sequence
3MM	Three matrix-matrix multiplications in sequence
AMX	Intel's Advanced Matrix Extension
AST	Abstract Syntax Tree
ATAX	Transposed matrix and Vector multiplication
AXPY	A times X plus Y
BICG	Sub kernel of BiCGStab (Biconjugate gradient stabilized method) linear solver
BLAS	Basic Linear Algebra Subprograms
CBLAS	C interface of BLAS
CDG	Control-Dependency Graph
CEGIS	Counter-Example Guided Inductive Synthesis
CIL	C MLIR frontend
CPU	Central Processing Unit
DDG	Data-Dependency Graph
DOT	Scalar product
DSL	Domain-Specific Language
EBNF	Extended Backus–Naur Form
FIR	Fortran MLIR frontend
FPGA	Field Programmable Gate Array
GEMM	GEneric Matrix Multiply
GEMV	GEneric Matrix-Vector multiplication
GPU	Graphics Processing Unit
HLS	High-Level Synthesis

IDL	Idiom Description Language
LLVM	The LLVM Compiler Infrastructure
ML	Machine Learning
MLIR	Multi-Level Intermediate Representation
MLT	Multi-Level Tactics
MVT	Two matrix-vector multiplications, with the second transposed
PAT	SMR pattern description format
PDLL	PDL Language
PFA	Pattern Finite Automaton
PGC	Pattern Generation Compiler
PGL	Pattern Generation Language
RISE	A functional pattern-based data-parallel language
SCAL	Scaling operation
SLY	Sly Lex-Yacc
SMR	Source Matching and Rewriting
SMT	Satisfiability Modulo Theories
SSA	Single Static Assignment
STNG	Verified lifting for stencil computations
SYMM	Symmetric matrix-matrix multiplication
SYR2K	Symmetric rank-2k update operation
SYRK	Symmetric rank-k update operation
TDL	Tactics Description Language
TDS	Tactics Description Specification
TWIG	A tree-manipulation language
VL	Verified Lifting

Contents

Chapter 1

Introduction

Writing a compiler optimization pass is a complex task that usually requires the intervention of a compiler expert. A subset of these optimizations is called *Idiom (or Pattern) Matching and Rewriting*, which consists of detecting a previously specified code idiom and replacing it with other optimized code fragment. Many applications can benefit from matching and rewriting optimization passes. For example, linear algebra program fragments can have their performance improved by replacing them with optimized library calls [6]. For example, GEneric Matrix Multiply (GEMM) is a relevant operation in ML models that can be accelerated by substituting an unoptimized implementation for a call to an optimized entry in the BLAS library [14].

An interesting application is to detect poor quality code during code reviews. Code standards that do not follow a certain quality level can be configured to issue alerts at compile time so as to facilitate software quality control. Another possible application is the detection of code patterns that are vulnerable to attacks. In such cases, the compiler can issue alerts to the programmer or replace the code with protected versions. In addition, idiom detection and replacement can be exploited as a way to speed up applications or to reduce the energy or memory consumption of applications' code fragments.

Tools to help specify pattern matching and rewriting optimizations have recently been proposed [8, 19, 10]. Their goal is to ease the task of specifying a pattern and its corresponding code replacement. Some of these tools use MLIR idioms as patterns [20] while others start from source code level patterns [11]. Source Matching and Rewriting (SMR) [8] is an optimizing compiler pass that uses a user-oriented source-code-based approach for MLIR idiom matching and rewriting. It does not require a compiler expert's intervention, as programmer-defined patterns are specified in the source language of the program to be compiled. However, increasing SMR pattern-matching coverage requires that the user describe all possible subtle variations of an idiom, a cumbersome and error-prone task. For example, consider the two code fragments in Listings 1.1 – 1.2 that implement a simple dot-product. Both perform the same operation (dot-product), differing only on how explicit the accumulation operation is specified. Since programmers can write many variations of such code, specifying all such variations is thus a tiresome task. Enabling programmers with a tool to specify patterns can considerably simplify such tasks.

This research introduces a Pattern Generation Language (PGL) design to ease the

```
1 for(int i=0;i<n;++i) {
2   out += x[i]*y[i];
3 }
```

Listing 1.1: Dot idiom using addition assignment (+=).

```
1 for(int i=0;i<n;++i) {
2   out = out + x[i]*y[i];
3 }
```

Listing 1.2: Dot idiom using separate assignment and addition.

task of automatically generating pattern variations. PGL is a high-level language that describes program patterns amenable to SMR matching and rewriting. Additionally, this work presents the PGL Compiler (PGC), an SMR-compatible code generation tool that: (a) automatically generates idiom variations; and (b) combines pattern variations to create new patterns. Although PGC is focused on generating SMR input patterns, it can also be modified to target other idiom-matching and rewriting tools.

The organizational structure of this work is as follows: Chapter 2 delves into the SMR pattern matching and rewriting approach and its execution flow. Chapter 3 describes all aspects of the PGL language, including its syntax and grammar, and shows how it generates pattern variations and combinations. The design and implementation of the PGC compiler are described in Chapter 3.2. Chapter 4 provides a set of experiments to show how PGL can expand SMR matching capabilities to increase patterns' application coverage. Research works similar to PGL are described and put in perspective in Chapter 5. Finally, Chapter 6 concludes the work and recommends future extensions.

Chapter 2

Background

This chapter starts by giving an overview of the MLIR operations in Section 2.1. The chapter concludes with a description on how the Source Matching and Rewriting (SMR) optimization works. It gives an overview of its inner workings and describes how patterns are captured and replaced by optimized code fragments at the MLIR-level representation of the compiler. To achieve that, it first discusses mechanics of SMR (Section 2.2) and how it generates Multi-Level IR (MLIR) fragments to replace code.

2.1 The Multi-Level IR (MLIR) Representation

One of the main challenges in building compilers is the need to support a wide range of programming languages and application domains. Different languages have different syntax, semantics, and optimization requirements, which can make it difficult to build a single compiler infrastructure that can handle them all. Additionally, different application domains have different performance requirements, which can make it difficult to optimize code for a wide range of hardware platforms.

Another challenge in building compilers is the need to support a wide range of hardware platforms. With the rise of heterogeneous computing, there is a growing demand for applications to support a wide range of hardware platforms, including CPUs, GPUs, FPGAs, and more. However, each of these platforms has different performance characteristics and programming models, which can make it difficult to optimize code for them.

MLIR [17] (Multi-Level Intermediate Representation) offers a versatile, multi-level structure that can be tailored for various programming languages, target architectures, or optimization domains. Within MLIR, customized representations are known as *dialects*. Notable examples include:

FIR Dialect Specifically designed for the Fortran programming language.

AMX Vector Dialect Utilized in Intel's Advanced Matrix Extension (AMX).

Affine Dialect Focused on linear algebraic operations.

A unique feature of MLIR is its ability to represent code from diverse sources, like programming languages' ASTs (Abstract Syntax Trees), HLS (High-Level Synthesis) circuits, and architecture-specific instructions. This versatility is exemplified in the Mojo

language, a Python superset that leverages MLIR's backend optimizations to achieve significant performance improvements without code rewriting.

In terms of structure, MLIR is composed of a graph where nodes are "Operations" and edges are "Values." Each operation can yield multiple results and accept several arguments. Operations encompass functions, function calls, and dialect-specific instructions.

MLIR's *Types* represent the data operated on by these operations. They include basic types like integers, floats, and tensors, and also support custom types. *Attributes* in MLIR are metadata elements associated with operations and types, containing information like constant values, annotations, and optimization-related properties.

Regions and *Blocks* are structural elements in MLIR. Regions group operations into logical units, like functions or loops, while Blocks organize operations within regions, akin to basic blocks in control flow constructs.

MLIR incorporates concepts from previous intermediate representations, such as LLVM. It employs SSA (Single Static Assignment) for reducing complexity and enhancing compiler optimizations. SSA ensures that each variable is defined only once before use.

```
"func"() ( {
^bb0(%arg0: !fir.ref<i32>, %arg1: !fir.ref<i32>):
  %c1_i32 = "std.constant"() {value = 1 : i32} : () -> i32
  %c0_i32 = "std.constant"() {value = 0 : i32} : () -> i32
  %0 = "fir.load"(%arg0) : (!fir.ref<i32>) -> i32
  %1 = "std.cmpi"(%0, %c1_i32) {predicate = 0 : i64} : (i32, i32) -> i1
  "fir.if"(%1) ( {
    "fir.store"(%c1_i32, %arg1) : (i32, !fir.ref<i32>) -> ()
    "fir.result"() : () -> ()
  },  {
    %2 = "fir.load"(%arg1) : (!fir.ref<i32>) -> i32
    %3 = "std.subi"(%2, %c1_i32) : (i32, i32) -> i32
    "fir.store"(%3, %arg1) : (i32, !fir.ref<i32>) -> ()
    %4 = "fir.load"(%arg1) : (!fir.ref<i32>) -> i32
    %5 = "std.cmpi"(%4, %c1_i32) {predicate = 0 : i64} : (i32, i32) ->
   i1
    "fir.if"(%5) ( {
      "fir.store"(%c0_i32, %arg0) : (i32, !fir.ref<i32>) -> ()
      "fir.result"() : () -> ()
    },  {
      "fir.result"() : () -> ()
    }) : (i1) -> ()
    "fir.result"() : () -> ()
  }) : (i1) -> ()
  "std.return"() : () -> ()
}) {sym_name = "_QPsum", type = (!fir.ref<i32>, !fir.ref<i32>) -> ()} :
   () -> ()
```

Listing 2.1: MLIR FIR dialect code example.

Furthermore, MLIR supports about 35 officially recognized dialects, including affine, gpu, llvm, vector, arith, omp, and tensor. These dialects extend MLIR's capability, allowing it to cater to a broad range of applications and hardware.

MLIR also uses *passes* for code transformation and optimization. It includes built-in passes for specific hardware platforms or application domains and allows for custom pass creation through a feature called *Interface*. This feature facilitates the development of generic optimizations relative to the operations performed.

The Multi-level lowering process allows code to be transformed in a way that is more transparent and understandable to developers. By breaking the lowering process into small steps, each of which can be inspected and understood independently, developers can more easily reason about the behavior of their code and diagnose performance issues.

Listing 2.1 shows an example of MLIR code in the FIR dialect, derived from Fortran. This code in Line 3 illustrates an operation from the 'std' dialect, named 'std.constant', which does not take operands and has a constant value of '1' of type 'int32', yielding a result of the same type.

2.2 Source Matching and Rewriting (SMR)

Source Matching and Rewriting (SMR) [8] is an optimizing compiler pass, developed within our research group, that uses a user-oriented source-code-based approach for MLIR idiom matching and rewriting. It does not require a compiler expert's intervention, as programmer-defined patterns are specified in the source language of the program to be compiled, such as *C* and *Fortran*.

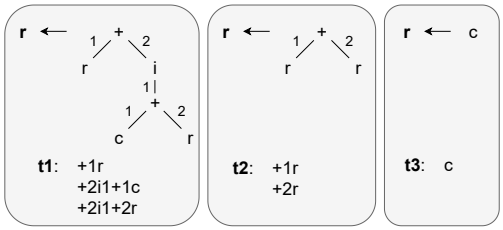


Figure 2.1: Tree Patterns.

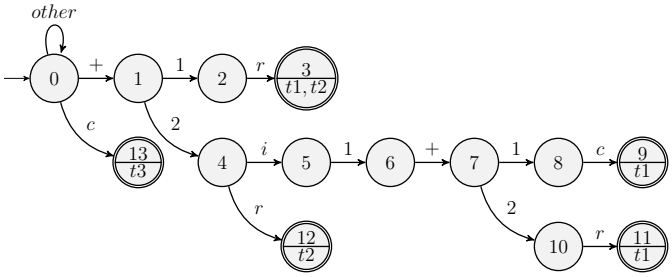


Figure 2.2: Tree Automaton.

It uses a 2-phase automaton to find information in a program's control graph and data graph. *SMR* builds on previous work [2] on *matching* patterns in trees (Figures 2.1 – 2.2) and uses the intermediate representation tool *MLIR*, which allows it to work with different source languages. [1]

Figures 2.1 – 2.2 present a brief overview of the operational mechanics of Twig's algorithms. Initially, in Figure 2.1(t1)-(t3) tree-patterns, undergo a process of linearization by mapping out all paths from the root to a leaf. This results in a set of path-strings, each uniquely representing a tree-pattern. Subsequently, these path-strings are translated into an Automaton, as illustrated in Figure 2.2. The automaton utilizes state transitions, annotated with path-string elements, and designates the last state of each path-string as final.

A noteworthy feature arises from the fact that different tree-patterns share common prefixes among their path-strings. Consequently, these path-strings traverse the same set

[1] Parts of the material below were adapted from [8].

of states in the automaton, contributing to a reduction in its overall size.

To exemplify, consider the path-string (+1r) corresponding to patterns t1 and t2 in Figure 2.1. In Figure 2.2, both path-strings traverse an identical sequence of states (0,1,2,3), culminating at state 3, marked as final. This state is annotated as having recognized the path string +1r from both tree-patterns t1 and t2. The criterion for a tree-pattern to match is that all its path-strings conclude in a final state.

Aho's [2] approach introduces a unified automaton-based representation for all patterns, yielding two notable advantages. Firstly, it conserves memory by merging common path-string prefixes, thereby reducing the overall size of the automaton. Secondly, the unified representation ensures that any input string is automatically compared against each path-string of every pattern, resulting in a significant reduction in execution time as the size and number of patterns increase. These advantages prove valuable when dealing with a substantial number of patterns in the matching process.

The key motivation behind SMR is to use it to leverage modern ISA extensions and hardware accelerators by detecting and raising program idioms to acceleration instructions or optimized library calls. While recent works based on MLIR have been proposed for code raising, they rely on specialized languages, compiler recompilation, or in-depth dialect knowledge. SMR, on the other hand, is designed to be user-friendly and accessible to a wider range of programmers.

```
1 <lang> {
2   fun <idiom_name>(<type_1> <arg_1>, <type_2> <arg_2>...){
3     <matching_code>
4   }
5 } = {
6   fun <idiom_name>(<type_1> <arg_1>, <type_2> <arg_2>...){
7     <replacement_code>
8   }
9 }
```

Listing 2.2: PAT, the pattern description format of SMR.

An example pattern definition that works as an input to *SMR* is shown in Listing 2.2. The pattern is described in a new format, called *PAT*. First, in line 1, there is the definition of the language in which the pattern was written. The first group of braces (lines 1 to 5) represents the pattern to be matched. The function definition in line 2 is not part of the pattern to be detected. It works only as a wrapper that links the pattern input variables (in sequence) with their proper substitutions at line 6. The pattern to be matched is in line 3. At lines 5 – 9 is another code in braces that represents the replacement code. As before in the case of the matching pattern function (line 2), it is worth noting that the function definition in line 6 is not part of the replacement pattern, working only as a wrapper to specify its input variables. The actual pattern that will replace that in line 3 is in line 7. The function wrappers in PAT help the compiler capture the pattern descriptions while linking the pattern's input variables together.

To better understand PAT consider, for example, the Fortran and C PAT descriptions of the sdot inner-product shown Listings 2.3 – 2.4. The patterns were designed to capture sdot and replace them for optimized BLAS calls. BLAS is an optimized library [29] that

can considerably speed up linear algebra kernels. The difference between both descriptions are in the language defined in line 1 (f90 vs. c), the way of defining a function (line 2), and the different ways of calling the BLAS library from C and Fortran. In C, the pattern types are defined inside the function arguments (line 2), and in Fortran, they are defined below (lines 3-5).

```
1  f90 {
2    subroutine dot(N,x,ix,y,iy,out)
3      real, dimension(*) :: x, y
4      real :: out
5      integer :: N, ix, iy
6      do i = 1, N
7        out = out + x(i*ix) * y(i*iy)
8      end do
9    end subroutine
10 } = {
11   subroutine dot(M,p,ip,q,iq,out)
12     real, dimension(*) :: p, q
13     real :: out
14     integer :: M, ip, iq
15     external :: sdot
16     out = out + sdot(M, p, ip, q, iq)
17   end subroutine
18 }
```

Listing 2.3: Find a dot product patterns and substituting with a *BLAS* call in Fortran using *SMR*.

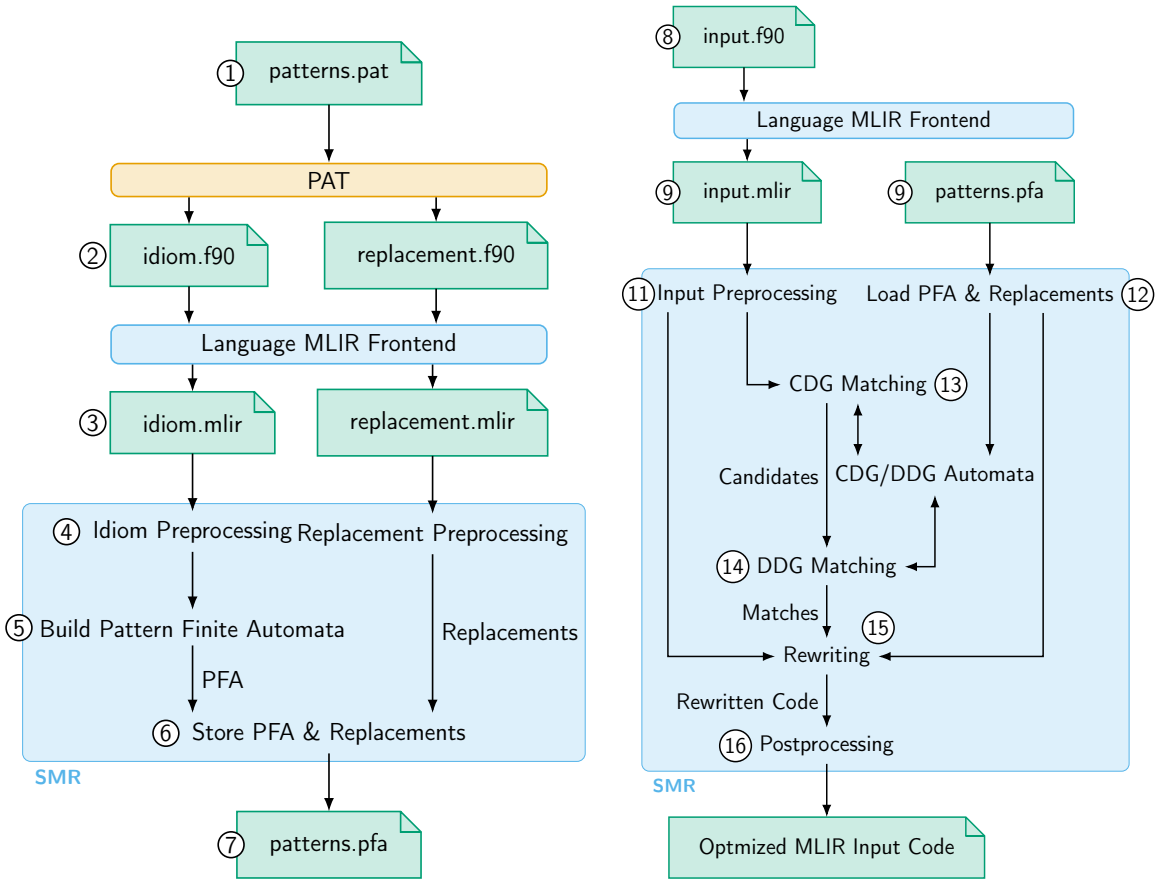
```
1  C {
2    void dot(int N,float *x,int ix,float *y,int iy,float out){
3      for (int i = 0; i  <  N;  i++)
4        out += x[i*ix] * y[i*iy];
5    }
6  } = {
7    #include <cblas.h>
8    void dot(int M,float *p,int ip,float *q,int iq,float out){
9      out += cblas_sdot(M, p, ip, q, iq);
10   }
11 }
```

Listing 2.4: Find a dot product patterns and substituting with a *BLAS* call in C using *SMR*.

The SMR algorithm runs in two processing phases. In the first phase, the idiom Control-Dependency Graph (CDG) is matched against the program's CDG to rule out code fragments that do not have a control-flow structure similar to the desired idiom. In the second phase, candidate code fragments from the previous phase have their Data-Dependency Graphs (DDGs) constructed and matched against the idiom DDG. This two-phase approach ensures that only relevant code fragments filtered by the first phase are considered for DDG matching, thus improving the efficiency and effectiveness of the algorithm.

SMR is effective in matching idioms from Fortran (FIR) and C (CIL) programs while raising them as BLAS calls to improve performance [8]. Additional experiments also show performance improvements when using SMR to enable code replacement in areas like approximate computing and hardware acceleration.

One problem with the SMR approach is that a programmer can typically write many source code variations of an idiom. To ensure the correctness of the optimization, the SMR matching algorithm is very strict and does not consider variations of each input idiom. This motivated the creation of an automatic generator of pattern variations, the Pattern Generation Compiler (*PGC*). The Pattern Generation Language (PGL) uses the PGL Compiler (PGC) to translate a pattern description written in PGL into a set of multiple patterns described in PAT, the input pattern description format for SMR.



(a) Pattern Finite Automaton Build flow with SMR [8].

(b) SMR [8] Input Matching/Rewriting flow.

Figure 2.3: The workflow of the SMR [8] tool.

The workflow of the SMR tool (Figure 2.3) can be divided into three main stages: pattern generation, pattern matching, and code rewriting.

In the first stage, the user provides a set of patterns ① that they want to match in some user application code ⑧. These patterns are expressed in the form of Control-Dependency Graphs (CDGs) and Data-Dependency Graphs (DDGs), which capture the control-flow and data-flow relationships between program statements. The CDGs and DDGs are then converted into path strings and used to generate a Pattern Finite Automaton (PFA) for

each pattern ⑤. At the SMR runtime, path strings from multiple patterns are coalesced to create a single Pattern Finite Automaton (PFA) file. This coalesced PFA file ⑦ is then used to match and rewrite the applications.

To speed up the process of matching, the automaton construction can be generated before the matching time. As shown in Figure 2.3a, the PAT description of idioms is converted individually to its MLIR representation, both idiom and replacement ③. A preprocessing ④ occurs to prepare the representation to be converted to an automaton. The Pattern Finite Automata is generated ⑤ and stored together with the replacements in a PFA file ⑥.

In the second stage, the PFA ⑨ for each pattern is used to match against the Control-Dependency Graph (CDG) ⑬ and Data-Dependency Graph (DDG) ⑭ of the input program ⑧. The matching algorithm is based on an automaton-based DAG-matching algorithm inspired by early work on tree-pattern matching [2]. The algorithm first matches the CDG of the input program against the CDG of the pattern to rule out code fragments that do not have a control-flow structure similar to the desired idiom. Then, candidate code fragments from the previous phase have their DDGs constructed and matched against the DDG of the pattern.

As shown in Figure 2.3b, with the automaton available in a PFA file ⑨, the user program input ⑧ is converted to its MLIR representation ⑨, the desired pattern finite automaton is loaded ⑫, and the input is preprocessed ⑪ Now the matching process starts with CDG Matching ⑬. The candidates are passed to the DDG Matching ⑭. If a match is identified, the idiom is rewritten appropriately ⑮ and postprocessed ⑯. At the end, the user will have the optimized MLIR input code, that can be compiled with the MLIR toolchain to an executable.

The rewriting process involves replacing the matched code fragments with optimized library calls or acceleration instructions. For example, if the pattern is a matrix multiplication, the matched code fragment can be replaced with a BLAS call to improve performance.

The automaton creation phase involves generating a Pattern Finite Automaton (PFA) for each pattern that the user wants to match in their code. The PFA is constructed by converting the Control-Dependency Graph (CDG) and Data-Dependency Graph (DDG) of the pattern into path strings. These path strings are then used to generate a finite automaton that can recognize the pattern in the input program. The PFA is stored in a file and loaded during the pattern-matching phase.

The CDG phase is the first step in the pattern-matching phase. In this phase, the CDG of the input program is matched against the CDG of the pattern to rule out code fragments that do not have a control-flow structure similar to the desired idiom. The CDG is a directed graph that represents the control-flow relationships between program statements. The CDG of the input program is constructed by analyzing the program's control-flow statements, such as if-else statements and loops. The CDG of the pattern is generated during the automaton creation phase.

The DDG phase is the second step in the pattern-matching phase. In this phase, candidate code fragments from the previous phase have their Data-Dependency Graphs (DDGs) constructed and matched against the DDG of the pattern. The DDG is a directed

graph that represents the data-flow relationships between program statements. The DDG of the input program is constructed by analyzing the program's data dependencies, such as variable assignments and function calls. The DDG of the pattern is generated during the automaton creation phase.

The CDG represents the control-flow relationships between program statements. It is a directed graph where each node represents a program statement, and each edge represents a control-flow relationship between two statements. For example, an if-else statement in a program would be represented by a node in the CDG, with edges connecting the node to the statements inside the if and else blocks. The CDG is used in the SMR algorithm to match the control-flow structure of a pattern against the control-flow structure of the input program.

The DDG, on the other hand, represents the data-flow relationships between program statements. It is a directed graph where each node represents a program statement that produces or consumes data, and each edge represents a data-flow relationship between two statements. For example, a variable assignment statement in a program would be represented by a node in the DDG, with edges connecting the node to the statements that use the variable. The DDG is used in the SMR algorithm to match the data dependencies of a pattern against the data dependencies of the input program.

Chapter 3

The Pattern Generation Language (PGL)

This chapter introduces the Pattern Generation Language (PGL), the main contribution of this dissertation. It provides a comprehensive definition and insights into PGL grammar and structure. The following sections delve into the motivations driving its design and provide a detailed overview of its key features.

PGL was designed to:

- be sufficiently flexible and powerful to enable the description of a variety of patterns;
- provide constructs that allow the expression of complex patterns;
- be easy to learn and use.

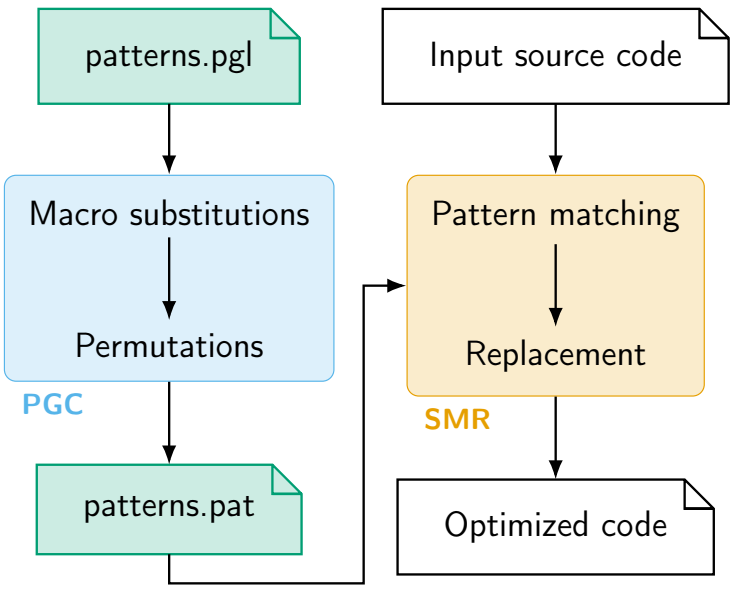


Figure 3.1: PGL Compiler integration flow with SMR [8].

An integrated workflow that combines PGL with SMR is shown in Figure 3.1. First, the pattern programmer writes in PGL all the desirable patterns that are stored in the

`patterns.pgl` file. Second, `patterns.pgl` is fed to the PGL Compiler (PGC) which code generates all variations of the programmed patterns storing them into the `patterns.pat` using the PAT format. Finally, `patterns.pat` is read by the SMR tool which performs pattern matching and replacement in the input (i.e. application) source code, as specified in the original PGL description, producing optimized code.

As described below, the expressiveness power of PGL allows pattern designers to easily describe a large and diverse set of idioms, thus increasing the chance of matching input program fragments. For example, in the previous chapter, it was shown that there are many ways to describe a dot product, specifically when programming the accumulation operation. As shown in (Listing 3.1) and detailed below, PGL addresses this problem by allowing the programmer to describe different variations of the accumulation operation without enumerating them. Although in the particular example of Listing 3.1 the programmer describes a C pattern, PGL is generic enough to enable pattern descriptions in any other language.

```
1  def inc(x) : ++x | x++ | x += 1 | x = x + 1
2  def init(x) : x = 0
3  def comp(a, b) : a < b
4  def for(x, y) : for ($init(x); $comp(x, y); $inc(x))
5  def acc(a, b) : a += b | a = b + a
6  def mul(a, b) : (a) * (b) | (b) * (a)
7  def vector(x, i) : x[i]
8
9  type int : unsigned int | int | unsigned long | long
10 type real : float | double
11
12 decl dot($int(n),$real(*x,*y,out)){
13   $init(out);
14   $for(i,n){
15     $acc(out,$mul($vector(x,i),$vector(y,i)));
16   }
17 } = {
18   #include <cblas.h>
19   $if($real==float){
20     out = cblas_sdot(n,x,1,y,1);
21   }
22   $else{
23     out = cblas_ddot(n,x,1,y,1);
24   }
25 }
```

Listing 3.1: DOT pattern written in PGL, using `float` and `double` types with the `$real` type expansion, to match a C idiom and replace with the `cblas` library.

Line 12 of Listing 3.1 declares the dot-product pattern and specifies its parameter types. The pattern is named `dot` and uses variables `n` of type `int`, and pointers `x`, `y`, and variable `out` all of type `real`. The type `int` was declared in Line 9 and represents both `signed`, `unsigned`, `int` or `long int` types from the C language. The type `real` was declared in Line 10 and represents both `float` or `double` types from the C language.

This way, the PGL compiler can synthesize dot-product pattern variations for both float and double data types and the 4 integer types.

Lines 1-7 represent the definitions constructs used later in the expansions. For example, the `acc` in Line 5 defines an accumulation operation with two variables, `a` and `b`. The multiple variations of this definition are separated by vertical bars and provide alternation between the code `a += b`, and `a = b + a`. Lines 13-16 of Listing 3.1 describe the code and expansions for the matching part of the pattern. Expansions in this context are identified by a dollar sign ($) followed by the expansion name and its arguments. The arguments provided are then substituted in sequence based on their corresponding order as defined in the expansion's definition. Lines 18-24 are the replacement part of the pattern. Line 18 is the library included for the BLAS call. Lines 19-24 represent the If operation, which compares the type of the `real` Type Expansion to choose different BLAS calls for each type.

3.1 PGL Syntax and Semantics

From the perspective of programming language design, a PGL (Pattern Generation Language) program fundamentally comprises four types of constructs: Definitions, Expansions, Conditionals, and Pattern Declarations. Listing 3.2 illustrates the structure of PGL, which serves as the pattern description format for PGC (Pattern Generation Compiler).

```
1  import "<def_file_location>"
2  def <def_name_1>(<arg_1>,...) : <variation_1> | <variation_2> | ...
3  def <def_name_2>(<arg_1>,<arg_2>,...) :  <variation_1> | <variation_2>
4  type <type_1> : <variation_1> | <variation_2> | ...
5  type <type_2> : <variation_1> | <variation_2>
6  decl <idiom_name>( <type_1>(<arg_1>,...),<type_2>(<arg_1>,<arg_2>,...) )
       {
7    <matching_code>
8    $<def_name_1>(<arg_1>,<arg_2>,...)
9    <matching_code>
10   $<def_name_2>(<arg_1>,<arg_2>,...)
11   ...
12   }
13 } = {
14   <replacement_code>
15   $if(<type_1> == <value>){
16     <replacement_code>
17   }
18   $elif(<type_2> != <value>){
19     <replacement_code>
20   }
21   $else{
22     <replacement_code>
23   }
24   <replacement_code>
25 }
```

Listing 3.2: PGL, the pattern description format of PGC.

In Listing 3.2, Line 1 imports definitions from an external file, which consists solely of 'def' and 'type' statements. Lines 2-3 present two example definitions using the 'def' keyword, specifying the definition name, argument names, and variations. Lines 4-5 introduce two example type definitions, employing the 'type' keyword followed by the type name and variations.

Lines 6-25 feature a pattern declaration. This begins with the 'decl' keyword and includes the pattern name, argument names with their corresponding types, a matching section (Lines 7-11), and a replacement section (Lines 14-24). The matching section (Lines 7-11) may contain code snippets in the target language (as seen in Lines 7,9) and definition expansions, denoted by the $ sign (Lines 8,10). These code snippets can intersperse expansions and conditionals in both the matching and replacement sections.

An expansion corresponds to a definition, and its arguments are replaced when the definition expands. Line 13, marked by an equal sign, signals the beginning of the replacement section (Lines 14-24). This section can also include multiple target language code snippets (Lines 14,16,19,22,24), with some embedded within a conditional block (Lines 15-23). The conditionals compare type variation values (defined in Lines 4-5) against constant values (as in Lines 15,18), allowing for varied replacement codes based on the type expanded in the matching section.

Definition

$$
\begin{aligned}
\langle \text{definition} \rangle &::= \texttt{"def"}\ \langle \text{ID} \rangle\ \texttt{"("}\ \langle \text{parameters} \rangle\ \texttt{")"}\ \texttt{":"}\ \langle \text{variations} \rangle \\
\langle \text{type\_definition} \rangle &::= \texttt{"type"}\ \langle \text{ID} \rangle\ \texttt{":"}\ \langle \text{variations} \rangle \\
\langle \text{parameters} \rangle &::= \langle \text{parameter} \rangle\ (\texttt{","}\ \langle \text{parameter} \rangle)* \\
\langle \text{variations} \rangle &::= \langle \text{instruction} \rangle\ (\texttt{"|"}\ \langle \text{instruction} \rangle)*
\end{aligned}
$$

In PGL, a 'definition' behaves similarly to a C-macro, providing a way to predefine operations for later use in PGL descriptions. As depicted in the presented PGL grammar fragment, a definition begins with the keyword 'def', followed by the definition's name. Parameters for the definition are then listed, enclosed in parentheses, and separated by commas. The syntax of a definition uses a colon to signify the beginning of the variations, which are delineated by vertical bars to indicate alternation between different instructions. A type definition begins with the keyword 'type', followed by the type's name. In this case, there is no parameters, only variations.

```
1 def inc(x): ++x | x++ | x += 1 | x = x + 1 | x = 1 + x
2 def sum(a, b): a + b | b + a
3 def mul(a, b): (a) * (b) | (b) * (a)
4 type int: int | unsigned int
5 type real: float | double
```

Listing 3.3: PGL Definition example.

Listing 3.3 showcases several examples of such definitions. For instance, line 1 in Listing 3.3 demonstrates a definition for various increments, a versatile operation commonly

used to update loop induction variables. Lines 2 and 3 of the same listing illustrate how PGL facilitates the definition of commutative variations for sum and multiplication patterns. Lines 4 and 5 demonstrates type definitions of integer and real types (float and double). Predefined sets of definitions can be seamlessly incorporated into a program using the 'import' keyword. This is followed by specifying the relative path to the definitions file. For instance, `import "fortran.def"` demonstrates how to import a file containing definitions. This feature is particularly beneficial for reusing a common set of definitions across multiple patterns, thereby streamlining the pattern-creation process and maintaining consistency.

Expansion

⟨expansion⟩ ::= `"$"` ⟨ID⟩ `"("` ⟨parameters⟩ `")"`
⟨type_expansion⟩ ::= `"$"` ⟨ID⟩
⟨parameters⟩ ::= ⟨parameter⟩ (`","` ⟨parameter⟩) ∗
⟨parameter⟩ ::= ⟨ID⟩
⟨expansion⟩

An Expansion in PGL involves replacing the contents of a definition, akin to invoking a C-macro. As shown in the grammar fragment above, it begins with a $ sign, followed by the definition ID and its actual variables enclosed within parentheses and separated by commas. A TypeExpansion begins with a $ sign followed by the type definition ID, without arguments. When an Expansion is utilized, each invocation triggers a permutation of the possible existing definitions, generating a resulting pattern for each combination. When a TypeExpansion is utilized, it is replaced by a fixed value for each resulting pattern. This way, the type generates consistent patterns that correspond to the function arguments.

A concrete example of an Expansion is depicted in Figure 3.2. In the example, a pattern written in PGL for sum and multiplication ⓪ undergoes expansion, resulting in eight variations (①-⑧) after considering all possible combinations. These variations encompass the two variations for the `sum` and `mul` definitions, along with two type variations derived from the 'real' keyword, namely `float` and `double`. As shown in the figure, one can estimate the total number of resulting variations as 2 (sum) × 2 (mul) × 2 (real) = 8.

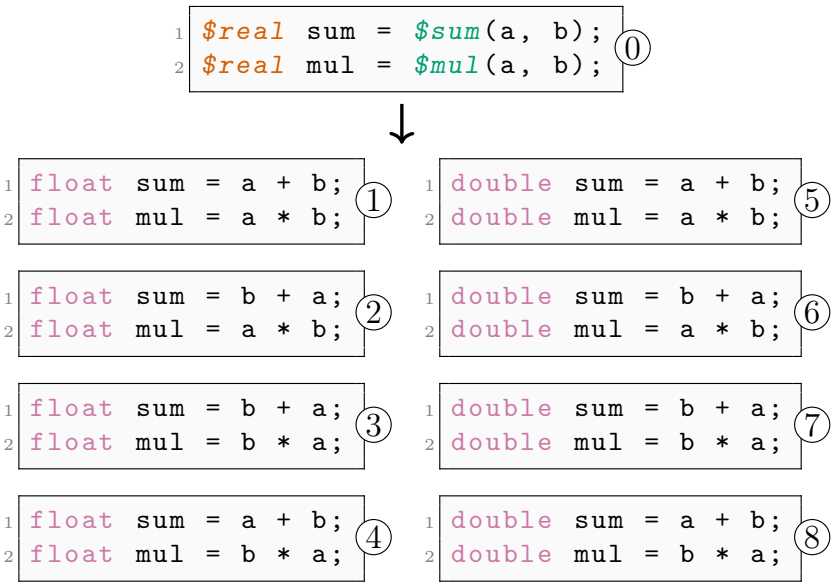


Figure 3.2: PGL Expansion example.

Conditional

⟨operation⟩ ::= ⟨condition_op⟩
⟨condition_op⟩ ::= `"$if"` `"("` ⟨condition⟩ `")"` `"{"` ⟨instruction⟩ ∗ `"}"` ⟨elif⟩ ∗ ⟨else⟩?
⟨condition⟩ ::= ⟨type_expansion⟩ ⟨operator⟩ ⟨ID⟩
⟨operator⟩ ::= `"=="`
`"!="`
⟨elif⟩ ::= `"$elif"` `"("` ⟨condition⟩ `")"` `"{"` ⟨instruction⟩ ∗ `"}"`
⟨else⟩ ::= `"$else"` `"{"` ⟨instruction⟩ ∗ `"}"`

In PGL, the conditional construct (*if-else*) plays a crucial role in influencing the pattern generation process during PGC compile time, based on specific conditions. As outlined in the grammar fragment, this construct is initiated with 'if', marked by a $ sign. The condition, enclosed in parentheses, typically comprises a type expansion (like '$real'), an operator (either equal or not equal), and an identifier (ID). At compile time, PGC evaluates this condition: if true, the patterns enclosed within the if-clause (denoted by braces) are generated; if false, the else-clause patterns are generated instead.

To add more flexibility, the construct includes optional keywords such as 'else' and 'elif'. The 'elif' keyword, combining the roles of `else` and an ensuing `if`, offers a seamless transition to additional conditions and can be used repeatedly. Conversely, the 'else' keyword, which does not require a condition, is used exclusively at the end of an if-else chain, and only once.

An example of the if-else construct is shown in the Figure 3.3. Depending on the type of the '$real' variable ⓪, the corresponding BLAS library function call is selected. If the type is float, a pattern with a call to the `cblas_sdot` is generated ①. Otherwise, if the type is `double`, a pattern with a call to the `cblas_ddot` is produced ②.

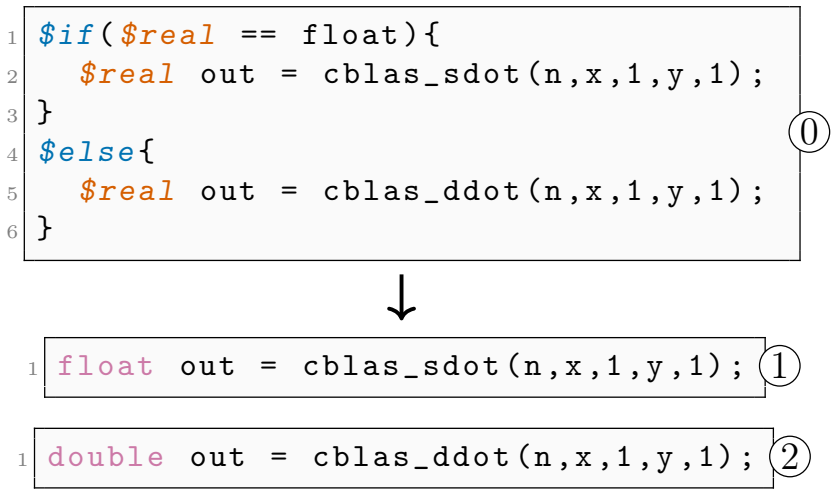


Figure 3.3: PGL Conditional Operation example.

Pattern Declaration

⟨pattern_decl⟩ ::= "decl" ⟨ID⟩ "(" ⟨parameters⟩ ")" ⟨block⟩ "=" ⟨block⟩
⟨block⟩ ::= "{" ⟨instruction⟩ ∗ "}"
⟨instruction⟩ ::= ⟨CODE⟩
⟨expansion⟩
⟨type_expansion⟩
⟨operation⟩

A Pattern Declaration is a PGL construct that defines new patterns based on definition and expansion constructs. It takes as input a set of pattern definitions, and its body (also called *pattern code*) uses expansions to program pattern variations. At its output, a pattern declaration synthesizes all possible patterns (in PAT format) that satisfy its PGL specification.

As shown in the grammar fragment above, the pattern declaration syntax is made up of a 'decl' keyword, followed by an ID with the name of the pattern, the definition of the pattern's input variables (in parentheses), and the matching code (in braces). This is followed by the `=` punctuation mark and another block to the replacement code (in braces).

An example of how the Pattern Declaration construct works is shown in Figure 3.4 for the C language. In the figure, the `sum2mul` PGL pattern is described in ⓪. That pattern takes as input the `$real` type definition (float or double) for variables `out`, `x`, and `y` (line 1) and replaces all possible statements resulting from the pattern variations produced by the expansion `$sum` (line 2) by statements that operate on the same arguments but use multiplication instead (line 4). After compiling the code in ⓪, the PGC compiler produces all possible sum2mul patterns resulting from the combinations of float and double input types (①–④). For example, the pattern variation in ① uses float for all input variables and expands the `$sum` as `x + y`. On the other hand, in pattern ①, all variables are

treated as double, and `$sum` is expanded to its commutative form `y + x`.

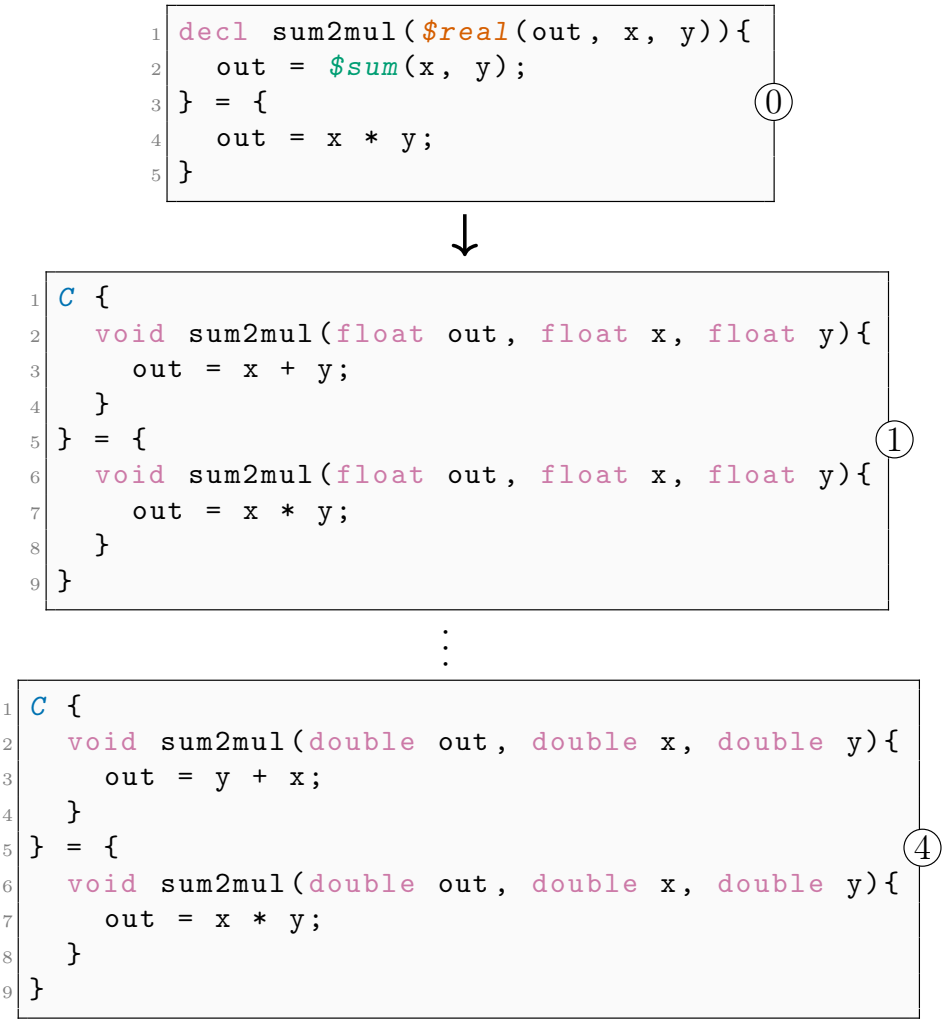


Figure 3.4: PGL Pattern Declaration example.

The GEMM Pattern

A more complex example of PGL programming is the Generic Matrix Multiply (GEMM) pattern shown in Listing 3.4. GEMM represents a fundamental idiom in linear algebra where two matrices are multiplied to produce a third matrix. An in-depth explanation of this idiom is found in the Section 4.1.1.

The definitions used by the GEMM pattern (Listing 3.4) are shown in lines 1-10. Lines 1 – 2 contain basic definitions like increment (`inc`), initialize (`init`), and compare (`comp`). A definition that can generate variations of simple for-loops is shown in line 4. The multiply and accumulate definition (`macc`) is in line 7, which uses expansions of definitions `acc` (line 5) and `mul` (line 6). Given that any GEMM pattern uses vector and matrix accumulation operations, the PGL program also includes definitions for them in lines 9 – 10. Type definitions are in lines 11 – 12. Notice that the `int` definition has only one variation, while the `real` type has variations `float` and `double`.

```
def inc(x): ++x | x++ | x += 1 | x = x + 1
def init(x): x = 0
def comp(a, b): a < b
def for(x, y): for ($init(x); $comp(x, y); $inc(x))
def acc(a, b): a += b | a = b + a
def mul(a, b): a * b | b * a
def macc(a, b, c): $acc(a,$mul(b,c))
def sum(a, b): a + b | b + a
def vector(x, i): x[i]
def matrix(A,b,c,d): $vector(A,$sum(b, $mul(c, d)))
type int: int
type real: float | double

decl gemm($int(m,n,k,lda,ldb,ldc),$real(alpha,*A,*B,beta,*C)){
  $int mm, nn, i;
  $for(mm,m){
    $for(nn,n){
      $real $init(c);
      $for(i,k){
        $real a = $matrix(A,mm,i,lda);
        $real b = $matrix(B,nn,i,ldb);
        $macc(c,a,b);
      }
    $matrix(C,mm,nn,ldc) = $sum($mul($matrix(C,mm,nn,ldc),beta),$mul(
   alpha,c));
    }
  }
} = {
  #include <cblas.h>
  $if($real == float){
    cblas_sgemm(m, n, k, alpha, A, lda, B, ldb, beta, C, ldc);
  }
  $elif($real == double){
    cblas_dgemm(m, n, k, alpha, A, lda, B, ldb, beta, C, ldc);
  }
}
```

Listing 3.4: GEMM pattern written in PGL.

The declaration for the GEMM pattern is in line 14. The indexes are declared using the type `int` and the array pointers and factors `alpha` and `beta` are declared using the type `real`. Lines 15-26, enclosed in braces, represent the desired matching part for the GEMM pattern. It consists of three nested for loops, using the `for` expansion. Lines 20-21 access the arrays at the desired position, and line 22 performs the multiply and accumulate using the `macc` expansion. Line 24 performs the final computation of addition and multiplications, storing at the output array `C`. The replacement for the detected GEMM patterns is surrounded by braces in lines 28-34. Line 28 includes the CBLAS library for the optimized function call. Lines 29-34 perform the selection of the appropriate BLAS function according to the `real` type. The function `cblas_sgemm` is called if the type is `float`. Otherwise, if the type is `double`, `cblas_dgemm` is called.

3.2 The PGL Compiler (PGC)

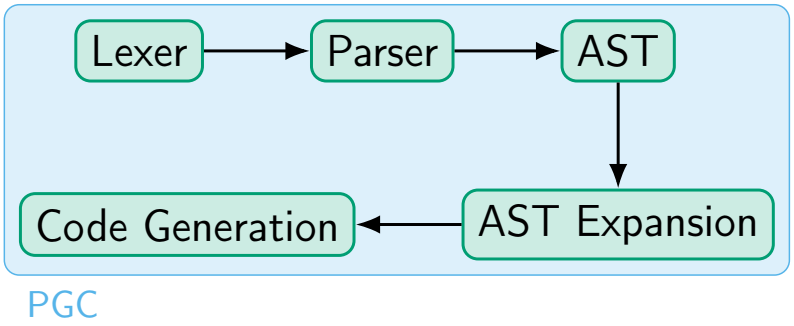


Figure 3.5: PGC Execution Flow.

The compilation flow of PGC is shown in the Figure 3.5. PGC was implemented in Python using the SLY[26] library. SLY provides two separate classes, Lexer and Parser. The Lexer class is used to break the pattern input written in PGL into a collection of tokens specified by a collection of regular expression rules. The Parser class is used to recognize the PGL language syntax. The two classes were combined to build a Lexer+Parser for PGC. As shown in Figure 3.5, an Abstract Syntax Tree (AST) for the PGL program is built after the execution of the lexical (Lexer) and syntactic (Parser) phases of the compiler. Finally, PGC traverses the AST to detect the expansion and pattern declaration constructs, using them to produce all pattern variations in the PAT format.

Lexer & Parser

The Pattern Generation Language (PGL) syntax is defined by the EBNF grammar in Listing 3.5.

From a syntax point of view, a program in PGL consists of:

- Imports
- Type Definitions

- Definitions (Function Definitions)
- Pattern Declarations

Imports are specified using the 'import' keyword followed by a `STRING`, containing the relative path to the definition file. Type Definitions are declared using the 'type' keyword, followed by an identifier (`ID`), a colon (`:`), and type variations. Definitions start with 'def' followed by an identifier (ID), parameters within parentheses, a colon (`:`), and variations of the definition that can be produced by expanding ID. Parameters are a list of identifiers (ID) separated by commas (`,`), which are used within expansions. Pattern Declarations begin with 'decl' followed by an identifier (ID) and parameters within parentheses. Expansions can be either simple identifier expansions (`$ID(parameters)`) or type expansions (`$ID`), both achieved through the expansion operator $. Variations consist of one or more instructions. Instructions can be actual code (CODE), expansions, type expansions, or conditional constructs.

⟨program⟩ ::= ⟨import⟩
⟨type_definition⟩
⟨definition⟩
⟨pattern_decl⟩

⟨import⟩ ::= `"import"` ⟨STRING⟩

⟨type_definition⟩ ::= `"type"` ⟨ID⟩ `":"` ⟨variations⟩

⟨definition⟩ ::= `"def"` ⟨ID⟩ `"("` ⟨parameters⟩ `")"` `":"` ⟨variations⟩

⟨parameters⟩ ::= ⟨parameter⟩ (`","` ⟨parameter⟩) ∗

⟨parameter⟩ ::= ⟨ID⟩
⟨expansion⟩

⟨variations⟩ ::= ⟨instruction⟩ (`"|"` ⟨instruction⟩) ∗

⟨pattern_decl⟩ ::= `"decl"` ⟨ID⟩ `"("`⟨parameters⟩`") {"` ⟨instruction⟩ ∗ `"} ="`
`"{"` ⟨instruction⟩ ∗ `"}"`

⟨expansion⟩ ::= `"$"` ⟨ID⟩ `"("` ⟨parameters⟩ `")"`

⟨type_expansion⟩ ::= `"$"` ⟨ID⟩

⟨instruction⟩ ::= ⟨CODE⟩
⟨expansion⟩
⟨type_expansion⟩
⟨operation⟩

⟨operation⟩ ::= ⟨condition_op⟩

⟨condition_op⟩ ::= `"$if"` `"("` ⟨condition⟩ `")"` `"{"` ⟨instruction⟩ ∗ `"}"`
⟨elif⟩ ∗ ⟨else⟩?

⟨condition⟩ ::= ⟨type_expansion⟩ ⟨operator⟩ ⟨ID⟩

⟨operator⟩ ::= `"=="`

```
                "!="
⟨elif⟩  ::=  "$elif" "(" ⟨condition⟩ ")" "{" ⟨instruction⟩ * "}"
⟨else⟩  ::=  "$else" "{" ⟨instruction⟩ * "}"
```

Listing 3.5: PGL EBNF grammar

The PGL grammar accepts programs written both in Fortran and in C, as it treats language operations as text and only differentiates specific PGL code by the use of the $ sign.

The generated parser tree is converted into an Abstract Syntax Tree (AST).

AST

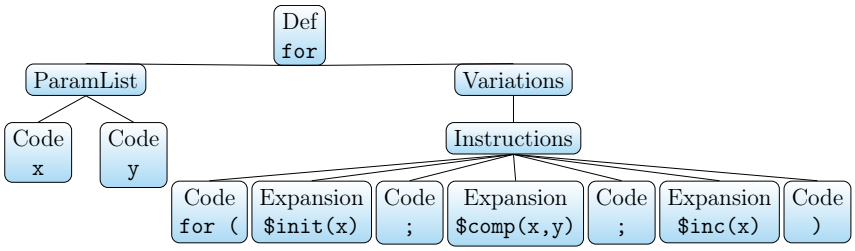
The creation of the PGL AST involves the utilization of Python classes and the combination of patterns through the itertools library [18]. The pertinent classes within the AST include Code, Instructions, ParamList, Lines, Variations, Block, Def, TypeExpansion, TypeDef, TypeDecl, PointerArray, Expansion, IfOperation, Comparison, and Declaration.

Following the construction phase, the AST undergoes a two-step process involving the expansion of definitions and a subsequent linearization procedure aimed at facilitating code generation. As an illustrative example, consider the `for` definition depicted in Figure 3.6. In Figure 3.6a, the initial AST resulting from parsing the `for` definition is displayed. The represented classes include Def (denoting a definition), ParamList (a list of parameters), Code (a segment of the source language code), Variations (elements subject to permutation), Instructions (a list comprising Codes, Variations, or Expansions), and Expansion (representing the expansion of a definition).

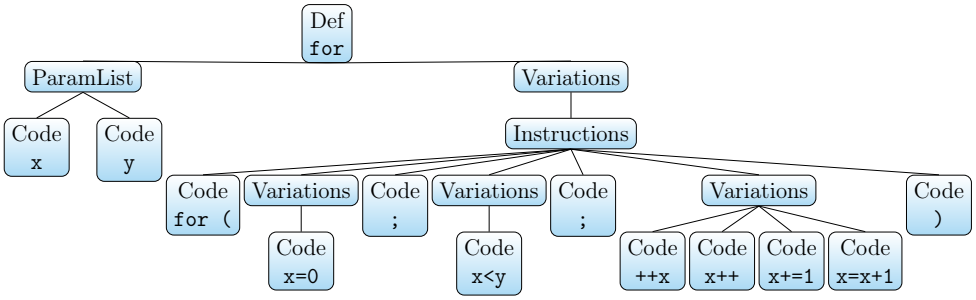
Figure 3.6b illustrates the detailed AST for the 'for' definition in PGL. In this figure, each 'Expansion' node from Figure 3.6a is substituted with its corresponding 'Variation', as defined in PGL. This substitution also includes updating the expansion parameters to align with the new context. For instance, the 'Expansion' node `$inc(x)` from Figure 3.6a is transformed in Figure 3.6b into a 'Variations' node with four child nodes. These child nodes represent different coding ways to increment 'x', specifically: `++x`, `x++`, `x+=1`, and `x=x+1`.

Finally, Figure 3.6c demonstrates the linearization of the AST. This process consolidates multiple Variation nodes into a single Variation node at a higher level within the AST. Achieved through the cartesian product of each lower-level Variation and their allocation to distinct branches within the higher-level Variation node.

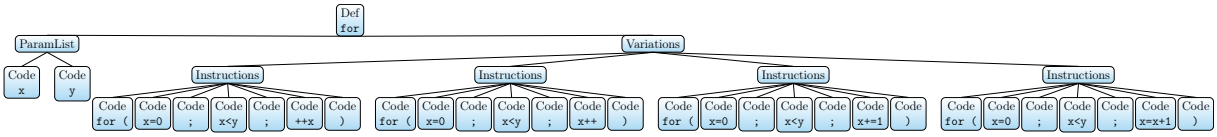
This pre-computation step significantly contributes to reducing the subsequent complexity of the code algorithm, which is responsible for generating all possible PAT variations for the `for` definition, as elaborated in the subsequent section.



(a) Initial AST of the `for` Definition.



(b) Expanded AST of the `for` Definition.



(c) Linearized AST of the `for` Definition.

Figure 3.6: PGC AST Expansion.

Code Generation

In the Code Generation phase of the PGL Compiler (PGC), output patterns in the PAT format are produced. This phase begins by processing the linearized Abstract Syntax Tree (AST), which includes nodes for Definitions (Def), Type Definitions (TypeDef), and Declarations.

Initially, the Pattern Declaration node is interpreted. Here, the 'decl' name within this node is converted into the name of the PAT function. Concurrently, the TypeDecl nodes are transformed into arguments for the resulting PAT pattern. The TypeDecl nodes include various combinations, all of which are generated using the 'itertools.product' function. This method results in a consistent and fixed set of combinations for each pattern, ensuring uniformity in type variations across all generated patterns. This structured approach is pivotal in maintaining consistency in pattern generation.

In the matching and replacement sections of the Pattern Declaration, the content of Expansion nodes is substituted using the Expansion dictionary. This substitution replaces the relevant variables with the arguments from the Expansion. Since the AST is already linearized and its sub-expansions are pre-combined, the substitution of Expansion nodes in the AST is efficiently executed.

The AST, now predominantly consisting of Code and Variations nodes, is next converted into a list-based format. In this format, each Variation becomes an element of a list, and Code is represented as a list containing a single element.

These list structures are then processed through the ‘itertools.product‘ function (cartesian product function), generating all required pattern combinations. The outcome of this process is formatted into text, resulting in the creation of the PAT patterns.

Chapter 4

Experimental Evaluation

This chapter details the experimental evaluation of the Pattern Generation Language (PGL), aimed at automating the generation of code pattern variations. The experiments focus on evaluating the usability, adaptability, strengths, and weaknesses of PGL, with a key finding being its enhanced pattern-matching quality compared to previous manual methods.

All experiments were performed using a dual Intel Xeon Silver 4208 CPU @ 2.10GHz with 16 cores total and 191 GiB of RAM running Ubuntu 20.04. As for the software toolchain, the following commits/versions have been used: (a) Flang (FIR) commit `8abd290` [24]; (b) CIL commit `195acc3` [23]; (c) LLVM/MLIR commit `1fdec59` [25]; (d) `OpenBLAS` version `0.3.20` [29]; and (e) GFortran version `9.4.0`.

4.1 Linear Algebra PGL Patterns

Central to the experiments were two carefully designed sets of PGL patterns, derived from Polybench and BLAS reference library, tailored for both Fortran and C languages, as detailed in Sections 4.1.1 – 4.1.2. The utilization of Polybench and BLAS kernels was a deliberate choice, given their widespread recognition as exemplary linear algebraic patterns, and their prevalent use across a variety of applications ranging from image processing to machine learning domains. These pattern sets were subjected to a rigorous evaluation process, focusing on aspects such as usability (Section 4.2.1), correctness (Section 4.2.2), coverage (Section 4.2.3), and variability (Section 4.2.4). The evaluation methodology involved converting these patterns into various formats using the PGL Compiler (PGC), followed by an in-depth analysis of the resultant outputs to assess their accuracy and practical effectiveness.

4.1.1 C PGL Patterns

To simplify the notation, Listing 4.1 lists all the definitions used by all the PGL descriptions in Section 4.1.1.

The six C patterns listed below were programmed in PGL. Following code generation with PGC, we conducted five measurements of the generation time and calculated the average. The quantity of pattern variations generated is detailed in Table 4.1.

```
def inc(x): ++x
def init(x): x = 0
def comp(a, b): a < b
def for(x, y): for ($init(x); $comp(x, y); $inc(x))
def acc(a, b): a += b | a = b + a
def mul(a, b): a * b | b * a
def macc(a, b, c): $acc(a,$mul(b,c))
def sum(a, b): a + b | b + a
def vector(x, i): x[i]
def matrix(A,b,c,d): $vector(A,$sum(b, $mul(c, d)))

type int : unsigned int | int | unsigned long | long
type real : float | double
```

Listing 4.1: PGL definitions used in C patterns.

When op(X) is used, it represents one of op(X) = X (same matrix) , or op(X) = A^T (transposed matrix).

Idioms	axpy	copy	dot	scal	gemm	gemv
# Patterns	160	40	160	136	86016	256
Generation time (ms)	62.5	21.58	66.18	57.89	26505.57	113.19

Table 4.1: Pattern variations generated with PGL for each idiom in C.

AXPY

The term "AXPY" stands for "a times x plus y". In this operation, you multiply a vector x by a scalar alpha, and then add the resulting vector to another vector y, as in the following equation:

$y = \alpha \times x + y$

An example of axpy idiom written in PGL uses the expansions `$for`, `$macc` and `$vector` and is presented in Listing 4.2.

```
decl axpy($int(n), $real(*x, *y, alpha)) {
  $int i;
  $for(i,n) {
    $macc($vector(y, i), $vector(x, i), alpha);
  }
}
```

Listing 4.2: AXPY pattern written in PGL.

COPY

The copy idiom is used to copy elements from one vector to another. It takes a vector x and produces a vector y, as in the following equation:

$y = x$

An example of a copy idiom written in PGL uses the expansions `$for` and `$vector` and is presented in Listing 4.3.

```
decl copy($int(n), $real(*x, *y)) {
  $int i;
  $for(i,n) {
    $vector(y, i) = $vector(x, i);
  }
}
```

Listing 4.3: COPY pattern written in PGL.

DOT

The dot product (also known as the scalar product or inner product) is an idiom that takes two vectors and produces a scalar. The dot product of two vectors, X and Y, is calculated as follows:

$out = \sum_{i=1}^{n} X_i Y_i$

An example of dot idiom written in PGL uses the expansions `$init`, `$for`, `$macc` and `$vector` and presented in Listing 4.4.

```
decl dot($int(n), $real(*x, *y, out)) {
  $init(out);
  $int i;
  $for(i,n) {
    $macc(out, $vector(x, i), $vector(y, i));
  }
}
```

Listing 4.4: DOT pattern written in PGL.

SCAL

The scaling idiom takes a vector and a scalar and produces a vector. The scaling operation of vector X with scalar alpha is calculated as follows:

$x = \alpha \times x$

An example of the scal idiom written in PGL uses the expansions `$for`, `$mul` and `$vector` and is presented in Listing 4.5.

```
decl scal($int(n), $real(*x, alpha)) {
  $int i;
  $for(i,n) {
    $vector(x, i) = $mul($vector(x, i),alpha);
  }
}
```

Listing 4.5: SCAL pattern written in PGL.

GEMV

The term GEMV stands for Generic Matrix-Vector multiplication. It represents a common idiom in numerical linear algebra where a matrix is multiplied by a vector. The matrix can be of any size, and the vector can be of any dimension, provided the number of columns in the matrix matches the dimension of the vector.

The gemv idiom multiplies a matrix A with a vector x and a scalar alpha, producing a result vector y that was previously scaled by beta. The operation is defined as follows:

$y = \alpha \times op(A) \times x + \beta \times y$

An example of the gemv idiom written in PGL uses the expansions `$for`, `$macc`, `$matrix` and `$vector` and is presented in Listing 4.6.

```
decl gemv($int(n),$real(*A,*X,*Y)) {
  $int i,j;
  $for(i,n) {
    $for(j,n) {
      $macc($vector(Y,i),$matrix(A,j,i,n),$vector(X,j));
    }
  }
}
```

Listing 4.6: GEMV pattern written in PGL.

GEMM

The term GEMM stands for Generic Matrix-Matrix Multiplication. It represents a fundamental idiom in linear algebra where two matrices are multiplied to produce a third matrix. The GEMM operation involves three matrices: A, B, and C. Matrices A and B are multiplied (A x B), and the result is scaled by alpha, producing alpha x A x B. C is then scaled by beta before the scaled matrix multiplication (alpha x A x B) is accumulated to it. This operation is defined as follows:

$C = \alpha \times op(A) \times op(B) + \beta \times C$

An example of the gemm idiom written in PGL uses the expansions `$for`, `$acc`, `$macc`, `$mul` and `$matrix` and is presented in Listing 4.7.

```
decl gemm($int(m,n,k,lda,ldb,ldc),$real(alpha,*A,*B,beta,*C)) {
  int mm, nn, i;
  $for(mm,m) {
    $for(nn,n) {
      $real $init(c);
      $for(i,k) {
        $macc(c,$matrix(A,mm,i,lda),$mul(alpha,$matrix(B,i,nn,ldb)));
      }
    $acc($matrix(C,nn,mm,ldc),c);
    }
  }
}
```

Listing 4.7: GEMM pattern written in PGL.

4.1.2 Fortran PGL Patterns

To simplify the notation, Listing 4.8 lists all the definitions used by all the PGL descriptions in Section 4.1.2.

```
def inc(x): ++x | x++ | x += 1 | x = x + 1
def init(x): x = 1
def comp(a, b): a < b
def for(x, y): do $init(x), y
def par(x): (x)
def acc(a, b): a = a + $par(b) | a = a + b
def mul(a, b): a * b | b * a
def macc(a, b, c): $acc(a,$mul(b,c))
def sum(a, b): a + b | b + a
def vector(x, i): x(i)
def matrix(x, i, j): x(i, j)

type int: integer
type real: real | double precision
```

Listing 4.8: PGL definitions used in Fortran patterns.

The nine Fortran patterns listed below were programmed in PGL. Following code generation with PGC, we conducted five measurements of the generation time and calculated the average. The quantity of pattern variations generated is detailed in Table 4.2.

Idioms	2mm	3mm	atax	bicg	gemm	mvt	symm	syr2k	syrk
# Patterns	32	8	32	32	32	16	4096	256	16
Gen. time (ms)	45.42	26.62	33.83	33.10	42.26	32.66	954.39	86.01	33.40

Table 4.2: Pattern variations generated with PGL for each idiom in Fortran.

2MM

The 2MM idiom is designed in PGL by performing two matrix-matrix multiplications in sequence, as defined below:

$D = alpha \times op(A) \times op(B) + D$

$E = op(C) \times op(D) + beta \times E$

The 2mm idiom is characterized by a sequence of two distinct Region Defining Operations (RDOs). Due to its structure, where each of the matrix multiplications performs a unique set of operations, it is not feasible to represent the 2mm idiom with a single PGL pattern. This limitation arises from the SMR's constraint of matching only one RDO per pattern. To effectively capture the essence of the 2mm idiom, it is necessary to devise two separate PGL patterns. Each pattern is designed to match one of the matrix-matrix multiplications: the first pattern accounts for the multiplication involving the alpha scalar, and the second handles the multiplication with the beta scalar. These patterns leverage the capabilities of `$for`, `$macc`, and `$mul` expansions in PGL, as detailed in the 2mm pattern representation in Listing 4.9.

```
decl p2mm1($int(ni,nj,nk),$real(alpha,a[nk][ni],b[nj][nk],tmp[nj][ni])){
  $for(i,ni)
    $for(j,nj)
      tmp(j,i) = 0.0
      $for(k,nk)
        $macc(tmp(j,i),alpha,$mul(a(k,i),b(j,k)))
      end do
    end do
  end do
}

decl p2mm2($int(ni,nj,nl),$real(beta,c[nl][nj],d[nl][ni],tmp[nj][ni])){
  $for(i,ni)
    $for(j,nl)
      d(j,i)=$mul(d(j,i),beta)
      $for(k,nj)
        $macc(d(j,i),tmp(k,i),c(j,k))
      end do
    end do
  end do
}
```

Listing 4.9: 2MM pattern written in PGL.

3MM

The 3MM idiom is designed in PGL by performing three matrix-matrix multiplications in sequence. The 3rd matrix-matrix multiplication uses the matrices resulting from the 1st and 2nd matrix-matrix multiplications.

$E = op(A) \times op(B) + E$

$F = op(C) \times op(D) + F$

$G = op(E) \times op(F) + G$

In contrast to the 2mm idiom's requirement for multiple PGL patterns, the 3mm idiom can be effectively represented using a single PGL pattern. This is possible because all three matrix-matrix multiplications in the 3mm idiom execute the same set of operations, allowing for a unified pattern description. The 3mm idiom effectively utilizes `$for` and `$macc` expansions within PGL, as illustrated in the detailed representation provided in Listing 4.10.

```
decl p3mm($int(ni,nj,nk),$real(a[nk][ni],b[nj][nk],e[nj][ni])){
  $for(i,ni)
    $for(j,nj)
      e(j,i) = 0.0
      $for(k,nk)
        $macc(e(j,i),a(k,i),b(j,k))
      end do
    end do
  end do
}
```

Listing 4.10: 3MM pattern written in PGL.

ATAX

The ATAX idiom is described as Matrix Transpose and Vector Multiplication and involves the following steps:

$tmp = A \times x$

$y = A^T \times tmp$

where A is the coefficient matrix, x is the input vector, tmp is the temporary vector, and y is the output vector. The A^T represents the transpose of matrix A.

This idiom uses the expansions `$for` and `$macc` as presented in Listing 4.11.

```
decl atax($int(nx,ny),$real(a[ny][nx],x[ny],y[ny],tmp[nx])){
  $for(i,nx)
    tmp(i) = 0.0D0
    $for(j,ny)
      $macc(tmp(i),a(j,i),x(j))
    end do
    $for(j,ny)
      $macc(y(j),a(j,i),tmp(i))
    end do
  end do
}
```

Listing 4.11: ATAX pattern written in PGL.

BICG

The BICG idiom in Polybench is the Sub Kernel of BiCGStab (Biconjugate gradient stabilized method) Linear Solver.

This idiom uses the expansions `$for` and `$macc` as presented in Listing 4.12.

```
decl bicg($int(nx,ny),$real(a[ny][nx],r[nx],q[nx],p[ny],s[ny])){
  $for(i,nx)
    q(i) = 0.0D0
    $for(j,ny)
      $macc(s(j),r(i),a(j, i))
      $macc(q(i),a(j, i),p(j))
    end do
  end do
}
```

Listing 4.12: BICG pattern written in PGL.

GEMM

The GEMM idiom performs the classic Generic Matrix Multiply (GEMM) operation as defined by the formula below: $C = alpha \times op(A) \times op(B) + beta \times C$

It uses the expansions `$for`, `$matrix`, `$mul` and `$macc` presented in Listing 4.13.

```
decl gemm($int(m,n,k),$real(alpha,beta,A[m][k],B[k][n],C[m][n])) {
    $for(nn,n)
    $for(mm,m)
      $matrix(c, mm, nn) = $mul($matrix(c, mm, nn), beta)
      $for(i,k)
        $macc($matrix(c, mm, nn),$mul(alpha,$matrix(b, i, nn)),$matrix(a
   , mm, i))
      end do
    end do
  end do
}
```

Listing 4.13: GEMM pattern written in PGL.

MVT

The MVT idiom comprises two matrix-vector multiplications, where the 2nd one has the A matrix transposed, as defined by the formulas below:

$x1 = A \times y1 + x1$

$x2 = A^T \times y2 + x2$

Both idioms use the expansions `$for` and `$macc` presented in Listing 4.14.

```
decl mvt1($int(n),$real(a[n][n],y[n],x[n])){
  $for(i,n)
    $for(j,n)
      $macc(x(i),a(i,j),y(j))
    end do
  end do
}

decl mvt2($int(n),$real(a[n][n],y[n],x[n])){
  $for(i,n)
    $for(j,n)
      $macc(x(i),a(j,i),y(j))
    end do
  end do
}
```

Listing 4.14: MVT pattern written in PGL.

SYMM

The SYMM idiom represents a symmetric matrix-matrix multiplication operation. Given a symmetric matrix A, a matrix B, and a matrix C, the SYMM pattern computes the following operation:

$C = \alpha \times A \times B + \beta \times C$

Here, A is a symmetric matrix, B is a matrix, C is the result matrix, α is a scalar coefficient, and β is another scalar coefficient.

This idiom uses the expansions `$for`, `$macc`, `$mul`, `$sum` and `$par` as presented in Listing 4.15.

```
decl symm($int(ni,nj),$real(acc,alpha,beta,a[nj][nj],b[nj][ni],c[nj][ni
  ])){
  $for(i,ni)
    $for(j,nj)
      acc = 0.0D0
        $for(k,j-2)
          $macc(c(j,k),$mul(alpha,a(i,k)),b(j,i))
          $macc(acc,b(j,k),a(i,k))
        end do
        c(j,i) = $sum($par($mul(beta,c(j,i))),$sum($par($mul(alpha,$mul(
  a(i,i),b(j,i))))),$par($mul(alpha,acc))))
    end do
  end do
}
```

Listing 4.15: SYMM pattern written in PGL.

SYRK

The SYRK pattern represents a symmetric rank-k update operation. Given a matrix A and a matrix B, the SYRK pattern computes the following operation:

$C = \alpha \times A \times A^T + \beta \times B$

Here, A is a matrix, C is the result matrix, α is a scalar coefficient, *beta* is another scalar coefficient, and A^T the transpose of matrix A.

This idiom uses the expansions `$for`, `$mul`, and `$macc` as presented in Listing 4.16.

```
decl syrk($int(ni,nj),$real(alpha,a[ni][ni],c[nj][ni])){
  $for(i,ni)
    $for(j,ni)
      $for(k,nj)
        $macc(c(j,i),alpha,$mul(a(k,i),a(k,j)))
      end do
    end do
  end do
}
```

Listing 4.16: SYRK pattern written in PGL.

SYR2K

The SYR2K pattern represents a symmetric rank-2k update operation. Given two matrices A and B, the SYR2K pattern computes the following operation:

$C = \alpha \times A \times B^T + \alpha \times B \times A^T + \beta \times C$

Here, A and B are symmetric matrices, C is the result matrix, α is a scalar coefficient, and β is another scalar coefficient. The $\times$ symbol represents matrix multiplication, and B^T and A^T represent the transpose of matrices B and A, respectively.

This idiom uses the expansions `$for`, `$mul` and `$macc` as presented in Listing 4.17.

```
decl syr2k($int(ni,nj),$real(alpha,beta,a[nj][nj],b[nj][ni],c[ni][ni])){
  $for(i,ni)
    $for(j,i)
      c(j, i)=$mul(c(j, i),beta)
    end do
    $for(j,i)
      $for(k,nj)
        $macc(c(j, i),alpha,$mul(a(k,i),b(k,j)))
        $macc(c(j, i),alpha,$mul(b(k,i),a(k,j)))
      end do
    end do
  end do
}
```

Listing 4.17: SYR2K pattern written in PGL.

4.2 Experimental Results

This section demonstrates the versatility of the Pattern Generation Language (PGL) in formulating patterns for different programming languages, specifically Fortran and C. The evaluation of PGL encompasses four critical aspects, each addressing a unique dimension of PGL as a pattern programming language:

1. **Usability** (Section 4.2.1): Here, the focus is on the practical application of PGL. Fortran patterns, created using PGL, were tested with the Source Matching and Rewriting (SMR) tool against the Polybench/Fortran suite. This test served to validate PGL's usability in real-world scenarios.

2. **Correctness** (Section 4.2.2): In this section, the emphasis is on the accuracy and reliability of PGL. C patterns generated through PGL were transformed into functions, compiled, and then executed against predetermined inputs. The outcomes of these tests were then compared to a reference standard to assess their correctness.

3. **Coverage Improvement** (Section 4.2.3): This part of the evaluation focused on PGL's efficiency in pattern matching. The study involved comparing the coverage achieved by automatically synthesized PGL C patterns against coverage from manually written PAT patterns, as described in [8]. This comparison aimed to demonstrate how PGL enhances pattern-matching capabilities.

4. **Pattern Variation Analysis** (Section 4.2.4): The final aspect of the evaluation explored the specific contributions of individual PGL pattern definitions to the overall pattern-matching process. This was carried out by examining the roles of 'accumulation' and 'multiply' definitions in pattern formation, providing a detailed case study in pattern variation analysis.

Each of these areas collectively underscores the comprehensive capabilities of PGL as a pattern programming language, highlighting its utility and effectiveness in various aspects of pattern generation and application.

4.2.1 Usability

In the first experiment, a set of patterns targeting Fortran BLAS kernels have been designed using the PGL language and applied to Fortran Polybench programs for matching.

To achieve that, first, the Polybench Fortran PGL patterns described in Section 4.1.2 were passed through the PGC compiler resulting in a set of PAT patterns that cover many variations of the input PGL patterns.

Second, the generated PAT patterns were used with SMR to run against the Polybench Fortran programs. Idioms were substituted by the corresponding Fortran BLAS calls.

Listings in Section 4.1.2 illustrates the PGL language straightforward representation for describing Polybench's kernel idioms, reiterating the usability of PGL with SMR.

With these experiments, it was possible to replicate all the matching results of the SMR paper [8] regarding the Fortran matching experiments. This achievement was accomplished by leveraging the PGL patterns outlined in Section 4.1.2. Notably, these PGL patterns offer a more versatile and flexible definition compared to the traditional PAT format.

4.2.2 Correctness Analysis

The goal of this set of experiments was to evaluate the correctness of the PGL language and the PGC compiler.

In order to validate the correctness of the generation of patterns and combinations, it was necessary to compile each generated pattern and execute them, comparing its output with an expected reference output. We validated all the patterns listed in Section 4.1.1 (axpy, copy, dot, scal, gemm and gemv) written in the C language. To test the pattern generation, it was necessary to extract only the match section of the patterns and insert them into functions of the language in which they were specified. These functions were divided into files with up to 10 functions and compiled in parallel with cmake and ninja. The main function instantiated the inputs (vectors or matrices) and saved the results in a variable or vector, using the first generated pattern as the reference pattern. After that, each pattern was executed with the same input values, and the output was compared with the reference output. This comparison takes into account that floating point operations have an error added to the result. Therefore, the accepted variation in this experiment was 10^{-5} for float and 10^{-14} for double. This validation program is compiled into a binary, that was produced for each type of pattern, performs the execution and comparison of all functions, and issues an error if the result is outside the threshold. With this experiment, it was possible to conclude that all generated pattern variations in Section 4.1.1 are equivalent.

4.2.3 Coverage Analysis

The goal of this section is to evaluate the ability of PGC to generate a broad range of pattern variations from a PGL description and to evaluate the coverage of these patterns on programs at large. To achieve that, the PGL patterns described in the Section 4.1.1 are passed through the PGC compiler, and the PAT patterns resulting from the PGC

output are fed to the SMR compiler. The SMR compiler takes as input a benchmark codebase and tries to pattern-match the PGC output patterns on them.

As stated in [4], the perfect evaluation scenario for pattern matching should consider matching patterns on a large-scale real-world codebase, but this brings many implementation problems as such code is not prepared for benchmarking. In addition, there is no known benchmark for pattern matching. To address that, this work uses AnghaBench [5], a large codebase containing one million programs randomly extracted from the Internet.

Although the goal of AnghaBench was not to evaluate pattern matching[1], it can still serve the purpose of this work. To achieve that, the AnghaBench programs were precompiled with the CIL MLIR frontend to filter only the compilable codes and then create a set of code snippets that work as a large codebase, on top of which the patterns generated by PGC from the PGL patterns in Section 4.1.1 could be matched.

Table 4.3, presents a comparison of pattern-matching results using PGL with SMR, focusing on different idioms across various programs like Darknet, Cello, Exploitdb, Ffmpeg, Hpgmg, and Nekrs. The table contrasts the number of idioms matched using SMR alone (Column A) and the combination of PGL with SMR (Column B). The baseline experiment and results were obtained from the coverage experiment of the SMR paper [8]. This comparison enables an evaluation of PGL's effectiveness in enhancing pattern-matching coverage. The data suggests that for certain programs and idioms, the use of PGL with SMR leads to a higher number of matches, indicating PGL's potential to expand pattern-matching capabilities. The total counts at the bottom of the table summarize the overall increase in pattern matches achieved by integrating PGL with SMR. The PGL result was inferior to that of SMR only in the case of Exploitdb. This discrepancy arose because the patterns utilized by SMR involved variations with global variables, a feature that was not implemented in the PGL language.

```
void axpy(int N, float alpha, float *x, int incx, float *y, int incy) {
  for(int i = 0; i < N; ++i) {
    y[i*incy] += alpha * x[i*incx];
  }
}
```

Listing 4.18: Pattern axpy not matched previously with SMR.

```
decl axpy($int(n, incx, incy), $real(*x, *y, alpha)) {
  $int i;
  $for(i,n) {
    $acc($vector_inc(y, i, incy), $mul($vector_inc(x, i, incx), alpha));
  }
}
```

Listing 4.19: Matched AXPY PGL pattern.

An illustration of PGL's capability for pattern matching is evident in the comparison of Listings 4.18 and 4.19. In Listing 4.18, an AXPY idiom failed to match in the SMR

[1] AnghaBench is used in predictive compilation to train compilers for code size reduction

```
void axpy(int n, int incx, int incy, float *x, float *y, float alpha){
  int i;
  for(i=0;i<n;++i) {
    y[i*incy]+=(x[i*incx])*(alpha) ;
  }
}
```

Listing 4.20: Matched AXPY PAT pattern generated with PGL.

experiment. However, with PGL's description of the same idiom, as shown in Listing 4.19, a successful match was achieved. The key difference between these pattern descriptions lies in the multiplication order of the scalar "alpha", highlighting how subtle variations in pattern description can significantly impact the matching process. The matched PAT pattern is detailed in Listing 4.20, demonstrating the effectiveness of PGL in capturing patterns that were previously unmatched.

Table 4.3: PGL with SMR matching results using CIL and CBLAS idioms.

	Darknet [22]		Cello [12]		Exploitdb [27]		Ffmpeg [7]		Hpgmg [1]		Nekrs [9]		**Total**	
Idiom	A	B	A	B	A	B	A	B	A	B	A	B	A	B
saxpy	1	2						1					1	3
scopy	1	1						9					1	10
sdot	1	3					1	4					2	7
sgemm	4	4											4	4
sscal	2	4						2					2	6
daxpy								1				3	0	4
ddot			1	1					1	2	2	2	4	5
dgemm					1						3	3	4	3
dgemv											1	1	1	1
dscal											3	4	3	4
Total	9	14	1	1	1	0	1	17	1	2	9	13	22	47

A: SMR.
B: PGL+SMR.

4.2.4 Pattern Variation Analysis

The goal of this section is to analyze how PGL can improve the matching of patterns that were impracticable before without the increasing flexibility of PGL definitions. The experiment starts by analyzing in greater depth the results of the experiments described in Section 4.2.3. In particular, it evaluates the impact of two central PGL definitions (`acc` and `mul`) in the matching result.

```
def acc(a, b) : a += b ① | a = b + a ②
def mul(a, b) : (a) * (b) ① | (b) * (a) ②
```

Listing 4.21: The definition of Acc and Mul written in PGL

The result of this evaluation is shown in Table 4.4. That table lists, for each input program, the PAT idioms that were matched from those produced after compiling the PGL patterns in Section 4.1.1. Columns `acc` and `mul` of Table 4.4 show which pattern variations have been matched at the input program based on the variations presented at the definitions in Listing 4.21. For example, line 1 Listing 4.21 describes a definition for `acc` in which two variations for addition are possible. Variation sentence `a += b` is marked ① while variation `a = b + a` is marked with ②. Similarly, in the definitions for `mul`, there are two variations: `(a) * (b)` marked with ① and its commutative form `(b) * (a)` marked with ②.

Variations marked with ① and ② in Listing 4.21 are associated with the columns `acc` and `mul` in Table 4.4, so an idiom matching can be broken down in which definition variations it used. For example, take idiom `scal` when matched inside the program Darknet. In the first matching of `scal` column 1 of both `acc` and `mul` are marked, meaning that idioms composed of the first variations of their corresponding definitions have been matched twice (see the last column). In addition, the second matching of `scal` was achieved by matching the first variation of `acc` and both variations (1 and 2) of `mul`.

Based on the Table 4.4, a deeper analysis reveals some insightful patterns and preferences in the use of 'acc' and 'mul' variations across different programs. The table indicates a tendency for certain variations to be more prevalent, suggesting that the way of writing a language in a program tends to repeat itself. This is evident in the frequent matching of the same patterns within different parts of the same input program. Such a trend implies that programmers often have a preferred way of implementing certain operations, which PGL can successfully capture and utilize for pattern matching. The dominance of variation ① in both 'acc' and 'mul' rules across multiple programs could reflect a more conventional or widely accepted way of writing these operations in code. This observation highlights the effectiveness of PGL in adapting to common coding practices and suggests potential areas for further optimization in pattern recognition algorithms, as the analysis suggests a tendency for only one type of variation to occur throughout the pattern.

Input	**Idiom**	`acc`		`mul`		**# Matches**
		①	②	①	②	
Cello [12]	dot	✓		✓		1
Cinder	copy	✓		✓		2
Darknet [22]	axpy	✓		✓		1
		✓	✓	✓	✓	1
	copy	✓		✓	✓	1
	dot	✓		✓		2
		✓	✓	✓	✓	1
	gemm	✓	✓	✓	✓	3
		✓	✓	✓		1
	scal	✓		✓		2
		✓		✓	✓	2
FFmpeg [7]	axpy	✓		✓		2
	copy	✓		✓		9
	dot	✓		✓		4
	scal	✓		✓		2
Hpgmg [1]	dot	✓		✓		2
Nekrs [9]	axpy	✓		✓		3
	dot	✓		✓		2
	scal	✓		✓		3
		✓		✓	✓	1
	gemm	✓	✓	✓	✓	3
	gemv	✓	✓	✓	✓	1
Numpy	scal	✓		✓		1
Petsc	dot	✓		✓		1
ReactOS	copy	✓		✓		1
RetroArch	axpy	✓		✓		1

Table 4.4: Ocurrence of `acc` and `mul` variations in idioms matched with PGL.

Chapter 5

Related Work

This chapter discusses key research relevant to our study, focusing on advanced techniques in programming language representation and optimization.

Shoaib Kamil et al. introduced *Verified Lifting* [13], a novel method for transforming low-level Fortran stencil computations into a high-level summary using a predicate language. This process is largely automated, relying on counter-example guided inductive synthesis (CEGIS) to ensure accurate and provably correct translations. Implemented within the STNG system, this approach yielded significant performance gains, with the translated code outperforming the original Fortran code by factors ranging from 4.1x to 24x.

The research also highlights the challenges in effectively using domain-specific languages (DSLs), pointing out the necessity of manual code rewriting. To address this, the authors developed *Metalift* [16], a new framework that automates the creation of DSL transpilers using program synthesis. Metalift, demonstrated to be efficient in translating and reducing code size, simplifies the complex process traditionally handled by compilers. Verified Lifting's extension into the Metalift framework is detailed, illustrating its use in pattern description through Python and efficient pattern identification using an SMT solver.

E-graphs, an innovation from the 1970s, are now instrumental in equality saturation for compiler optimizations and program synthesis. In the paper "Egg: Fast and Extensible Equality Saturation" [28], Max Willsey et al. present two methods to enhance e-graphs' efficiency and extensibility in this context. They also introduce "Ruler", a pattern rewriting tool that utilizes Egg and an SMT solver for equality saturation, offering a distinct approach compared to automaton-based methods like "SMR".

The paper "Rewrite Rule Inference Using Equality Saturation" [21] explores using e-graphs and equality saturation to infer rewrite rules in compilers, synthesizers, and theorem provers. Rewrite rules, crucial for simplifying expressions or establishing equivalences, are often challenging to develop. The authors demonstrate how equality saturation can efficiently generate smaller, more adaptable rulesets, highlighting "Ruler" for its ability to synthesize rulesets 25 times faster than similar tools without losing proving power. In a case study, Ruler-generated rules performed comparably to expert-crafted rules and solved issues in an open-source tool.

"RISE" [19] is a parallel data functional language integrated with MLIR, comple-

mented by "Elevate" [11], a language for describing optimization strategies. These tools, successors to the "Lift" project [15], feature a pattern detection system and device-specific optimizations. RISE primarily utilizes parallel data operators like "map" and "reduce".

"PDL" [20] developed by the MLIR community, is a language designed for pattern replacement within dialects, independent of external tools. It adopts a pattern-matching approach similar to "SMR" but focuses on defining patterns directly in dialects, whereas SMR uses source languages like C and Fortran.

Polyhedral and BLAS optimizations are implemented using the "Tactics Description Language" from "Multi-Level Tactics" (MLT) [3], which is based on MLIR. Descriptions in TDL are converted into "Tactics Description Specification," using MLIR's "TableGen". However, this support is limited to high-level operations such as reshape, transpose, matmul, matvec, and convolution.

"Idiom Description Language" (IDL) [10] is used for describing substitution patterns for LLVM, mapping patterns to calls of optimized libraries and domain-specific languages like Halide and Lift [15]. One challenge with this approach is pattern recognition post-LLVM lowering, as code distortions can reduce pattern detection. Using MLIR mitigates this by enabling higher-level matching, thus preserving more of the source language's structure.

Table 5.1 compares the programs cited above, like Verified Lifting (VL), Metalift, Ruler, Elevate, PDL, Multi-Level Tactics (MLT), and Idiom Description Language (IDL) against key features: MLIR, LLVM, Automaton, SMT Solver, and TableGen. Verified Lifting and Metalift are both automated systems that utilize SMT Solvers for high-level abstraction and efficient pattern recognition. Ruler stands out for its use of SMT Solvers, focusing on equality saturation and rewrite rule inference. Elevate, closely integrated with MLIR, emphasizes optimization strategies without incorporating LLVM or SMT Solvers. PDL, uniquely, is tailored to MLIR with a specific emphasis on pattern replacement within dialects. MLT employs both MLIR and TableGen for optimizing complex operations like polyhedral and BLAS optimizations. IDL, conversely, is more aligned with LLVM, using it for pattern description in low-level code.

In contrast, our work with PGL+SMR is distinct for its feature of Source Language Pattern Description. While other programs in the table focus on either high-level abstraction (like in VL and Metalift) or low-level code optimization (such as in Ruler and IDL), PGL+SMR stands out for its ability to describe patterns directly in the source language. This unique feature allows for more intuitive and accessible pattern recognition and manipulation, bridging the gap between high-level conceptual understanding and low-level code optimization. This capability makes PGL+SMR particularly versatile and user-friendly, especially for developers who may not be deeply versed in MLIR or LLVM intricacies. Overall, while each program has its strengths, PGL+SMR's focus on source language pattern description offers a distinctive and practical approach in the realm of programming language representation and optimization.

Features	**Programs**							
	VL	Metalift	Ruler	Elevate	PDL	MLT	IDL	PGL+SMR
MLIR	✗	✗	✗	✓	✓	✓	✗	✓
LLVM	✗	✗	✗	✗	✗	✗	✓	✗
Automaton	✗	✗	✗	✓	✓	✗	✗	✓
SMT Solver	✓	✓	✓	✗	✗	✗	✗	✗
TableGen (graph)	✗	✗	✗	✗	✗	✓	✓	✗
Languages	Fortran	Java	✗	Rise	MLIR IR	MLIR IR	LLVM IR	C/Fortran
Source language pattern description	✗	✗	✗	✓	✗	✗	✗	✓

Table 5.1: PGL and Related Works feature comparison.

Chapter 6

Conclusions and Future Work

This dissertation introduces and evaluates the Pattern Generation Language (PGL), an innovative tool designed to simplify and enhance the process of code detection and rewriting. PGL enables users to describe complex program patterns, which are then processed by the PGL Compiler (PGC) to automatically generate a variety of pattern variations. This research also explores the integration of PGL with existing Source Matching and Rewriting (SMR) and Multi-Level Intermediate Representation (MLIR) tools.

Our experimental analysis demonstrates PGL's capability to significantly increase the number of pattern matches in comparison to manually written patterns for SMR. This enhancement is particularly notable in its application to real-world programs, where PGL's ability to generate diverse pattern variations leads to more effective code optimization. The study also identifies several pattern characteristics that contribute to improved matching efficiency.

While PGL has shown considerable promise in this domain, future research avenues remain open:

Parallel Code Generation Enhance PGL's code generation process with parallel computing techniques. This could involve optimizing the PGL Compiler to exploit parallelism, thereby accelerating the generation of pattern variations and improving overall performance.

Cross-Language Compatibility Extend PGL's capabilities to support a wider range of programming languages. Investigate methods to make PGL patterns more language-agnostic, enabling seamless application across diverse codebases written in different languages.

Semantic Analysis for Code Reordering Enhance PGL with advanced semantic analysis capabilities to automatically identify opportunities for code reordering. This could involve analyzing dependencies and interactions between instructions to improve code efficiency further.

Pattern Repository and Sharing Develop a centralized repository for PGL patterns, allowing users to share and reuse patterns across different projects. This collaborative approach could foster a community-driven ecosystem for pattern development and optimization strategies.

Energy Efficiency Optimization Investigate PGL's potential in optimizing code for energy efficiency. Develop patterns and strategies that consider energy consumption metrics, contributing to the broader goal of sustainable computing.

By addressing these areas, we aim to broaden PGL's applicability and efficiency in the field of compiler optimization, particularly in the realms of code quality control, security enhancement, and performance improvement in various computing applications.

Bibliography

[1] Mark Adams, Jed Brown, John Shalf, Brian Van Straalen, Erich Strohmaier, and Sam Williams. HPGMG 1.0: A benchmark for ranking high performance computing systems. Technical Report LBNL-6630E, University of California, Lawrence Berkeley National Laboratory, 6 2014. URL: https://escholarship.org/uc/item/00r9w79m.

[2] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst.*, 11(4):491–516, oct 1989. doi:10.1145/69558.75700.

[3] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. Progressive raising in multi-level IR. pages 15–26. IEEE, 2 2021. doi:10.1109/CGO51591.2021.9370332.

[4] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 86–99, 2017. doi:10.1109/CGO.2017.7863731.

[5] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quinão Pereira. ANGHABENCH: A suite with one million compilable C benchmarks for code-size reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 378–390, 2021. doi:10.1109/CGO51591.2021.9370322.

[6] J. Demmel. LAPACK: a portable linear algebra library for supercomputers. In *IEEE Control Systems Society Workshop on Computer-Aided Control System Design*, pages 1–7, 1989. doi:10.1109/CACSD.1989.69824.

[7] FFmpeg Developers. FFmpeg tool, 2023. URL: https://ffmpeg.org/.

[8] Vinicius Espindola, Luciano Zago, Hervé Yviquel, and Guido Araujo. Source matching and rewriting for MLIR using string-based automata. *ACM Trans. Archit. Code Optim.*, 20(2), mar 2023. doi:10.1145/3571283.

[9] Paul Fischer, Stefan Kerkemeier, Misun Min, Yu-Hsiang Lan, Malachi Phillips, Thilina Rathnayake, Elia Merzari, Ananias Tomboulides, Ali Karakus, Noel

Chalmers, and Tim Warburton. NekRS, a GPU-accelerated spectral element Navier–Stokes solver. *Parallel Computing*, 114:102982, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0167819122000710, doi:10.1016/j.parco.2022.102982.

[10] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. Automatic matching of legacy code to heterogeneous APIs. pages 139–153. ACM, 3 2018. doi:10.1145/3173162.3173182.

[11] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Sergei Gorlatch, and Michel Steuwer. A language for describing optimization strategies. *CoRR*, abs/2002.02268, 2020. URL: https://arxiv.org/abs/2002.02268, arXiv:2002.02268.

[12] Daniel Holden. Cello: Higher level programming in C, 2015. URL: https://libcello.org/.

[13] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. pages 711–726. ACM, 6 2016. doi:10.1145/2908080.2908117.

[14] Bo Kågström, Per Ling, and Charles van Loan. GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw.*, 24(3):268–302, sep 1998. doi:10.1145/292395.292412.

[15] Martin Kristien, Bruno Bodin, Michel Steuwer, and Christophe Dubach. High-level synthesis of functional patterns with Lift. pages 35–45. ACM, 6 2019. doi:10.1145/3315454.3329957.

[16] Shadaj Laddad, Sahil Bhatia, and Alvin Cheung. Metalift, July 2023. URL: https://metalift.pages.dev/.

[17] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. pages 2–14, 2021. doi:10.1109/CGO51591.2021.9370308.

[18] Itertools Python library. Documentation, 2023. URL: https://docs.python.org/3/library/itertools.html.

[19] Martin Lücke, Michel Steuwer, and Aaron Smith. Integrating a functional pattern-based IR into MLIR. pages 12–22, 2021. doi:10.1145/3446804.3446844.

[20] MLIR. PDL language, July 2023. URL: https://mlir.llvm.org/docs/PDLL/.

[21] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. Rewrite rule inference using equality saturation. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021. doi:10.1145/3485496.

[22] Joseph Redmon. Darknet: Open source neural networks in C, 2016. URL: http://pjreddie.com/darknet/.

[23] CIL repository. Commit, 2021. URL: https://github.com/compiler-tree-technologies/cil/commit/195acc33e14715da9f5a1746b489814d56c015f7.

[24] FIR repository. Commit, 2021. URL: https://github.com/flang-compiler/f18-llvm-project/commit/8abd290c2c791c26cd1237b218def1b85998d403.

[25] LLVM/MLIR repository. Commit, 2021. URL: https://github.com/llvm/llvm-project/commit/1fdec59bffc11ae37eb51a1b9869f0696bfd5312.

[26] SLY repository. Commit, 2023. URL: https://github.com/dabeaz/sly/commit/33d4f5afc04c9cae78bd7dc094e08badc738409e.

[27] Offensive Security. The official exploit database repository, 2023. URL: https://github.com/offensive-security/exploitdb.

[28] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi:10.1145/3434304.

[29] Zhang Xianyi and Martin Kroeker. OpenBLAS: An optimized BLAS library, 2020. URL: https://www.openblas.net/.