Trade-offs in Generic Programming: A Cross-Language Performance Study

by

Daniel Pang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Masters of Math
in
Computer Science

Waterloo, Ontario, Canada, 2026

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Marc Moreno Maza Professor, Dept. of Computer Science, University of Western Ontario
Internal Member:	Éric Schost Professor, Cheriton School of Computer Science, University of Waterloo
Supervisor:	Stephen M. Watt Professor, Cheriton School of Computer Science, University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Generic programming enables abstraction and code reuse, but its impact on performance can vary significantly depending on language design and implementation strategy. Although modern programming languages differ substantially in compilation model, runtime system, and type system design, many share a common foundation of parametric abstraction and constrained generic programming, allowing scientific algorithms to be expressed in a largely portable manner across platforms.

This work investigates the trade-offs between generic and specialized implementations across multiple programming languages, with a focus on how generic realization strategies interact with runtime representation and compiler behaviour to determine performance characteristics. To evaluate these trade-offs, several arithmetic-intensive algorithms, including numerical and symbolic kernels, as well as a symbolic Gröbner basis computation, were implemented in both generic and specialized forms across a selected set of languages. These implementations were benchmarked to measure performance differences while also considering factors such as binary size and development experience.

The results show that the cost of generic programming is not inherent to abstraction itself, but instead depends on how generic abstractions are realized and optimized by the language and compiler. Languages employing compile-time specialization through monomorphization, such as C++ and Rust, often approached the performance of specialized implementations in the evaluated workloads, while approaches relying on boxed representations or runtime polymorphism often exhibited measurable overhead associated with dynamic dispatch, allocation, and additional levels of indirection. Runtime specialization approaches occupy an intermediate position, trading predictable compilation behaviour for adaptive optimization.

Overall, this work demonstrates that the relationship between abstraction and performance is shaped by language design, compiler strategy, and application context. These findings provide insight into both the shared foundations and differing realization strategies of modern generic systems, offering guidance for practitioners and language designers evaluating generics in performance-critical computing environments.

Acknowledgements

I express my gratitude to my supervisor, Stephen M. Watt, for his guidance and support throughout and beyond this research. I also thank the members of my examining committee for their valuable feedback.

I am also grateful to the University of Waterloo for providing a system and environment for me to learn and grow in the process of conducting this research.

Finally, I would like to express my appreciation to my friends and family for their encouragement and support during this time.

Dedication

To my parents. My appreciation for your support and understanding is bottomless. Thank you both.

Table of Contents

List of Tables

Chapter 1

Introduction

Scientific computing plays a central role in modern research and engineering, enabling the simulation, analysis, and optimization of complex systems across domains including physics, engineering, and data science. These applications are often computationally intensive, relying on highly optimized numerical algorithms to achieve practical performance. As a result, programming languages used in scientific computing are frequently evaluated not only on their expressiveness and ease of use, but also on their ability to produce efficient executable code.

At the same time, modern programming languages increasingly emphasize abstraction mechanisms that improve code reuse and maintainability. One such mechanism is *parametric polymorphism*, commonly referred to as generics. Generics allow algorithms to be written independently of the specific data types on which they operate, enabling a single implementation to be reused across multiple domains. In scientific computing, this abstraction is particularly attractive, as many algorithms can be expressed over general algebraic structures such as fields or rings rather than concrete numeric types.

The interaction between abstraction and performance, however, is not uniform across programming languages. While generics provide clear benefits in modularity and expressiveness, their runtime cost depends critically on how they are realized by the language and its compiler. Different languages adopt fundamentally different realization strategies for generics, including compile-time specialization (monomorphization), type erasure, and runtime specialization. These strategies determine when type information is resolved and to what extent generic code can be optimized, leading to varying trade-offs in runtime performance, compilation cost, binary size, and predictability.

Despite these differences, the evaluated languages share a common foundation of para-

metric abstraction, constrained interfaces, and reusable type-independent algorithm design. Many scientific computing kernels can therefore be expressed in a largely portable generic form across substantially varying programming environments. The primary differences arise not in the ability to express generic abstractions themselves, but in how those abstractions are realized, specialized, and represented at runtime.

Rather than admitting a single optimal solution, these design choices reflect a spectrum of trade-offs. Compile-time specialization can produce highly efficient code at the cost of increased compilation time and larger binaries, while type erasure reduces code duplication but may introduce indirection and limit optimization opportunities. Runtime specialization can adapt to dynamic workloads but introduces additional complexity and warm-up overheads. The impact of these trade-offs is further shaped by the application context, as tightly optimized numerical kernels, reusable library code, and irregular symbolic computations place different demands on abstraction mechanisms.

Previous work has explored the performance implications of generics in scientific computing, most notably through adaptations of the SciMark benchmark suite and the development of SciGMark. These studies demonstrated that the cost of abstraction varies widely depending on language design, with compile-time specialization often achieving performance comparable to hand-written specialized code, while runtime-based approaches may incur measurable overhead. However, much of this prior work is limited to older programming languages and primarily focuses on numerical workloads. Since then, programming languages and compiler technologies have evolved significantly, introducing new models for generics and new trade-offs between performance, compilation time, and code size.

This thesis revisits and extends this line of research through a cross-language evaluation of generic programming in modern programming environments. The study compares *specialized* and *generic* implementations of a shared set of computational kernels across multiple programming languages, including Rust, Java, Go, TypeScript, Julia, and C++. These languages were selected to represent a diverse range of design choices, including ahead-of-time and just-in-time compilation, manual and automatic memory management, and distinct approaches to the realization of generics.

The primary objective of this work is not to identify a universally superior approach to generics, but to characterize how alternative realization strategies trade runtime performance, compilation behaviour, binary size, and abstraction cost across different classes of applications. To achieve this, a suite of benchmark algorithms is implemented in both specialized and generic forms. The numerical component of the benchmark suite is derived from SciMark and includes kernels such as Fast Fourier Transform, Successive Over-Relaxation, Monte Carlo integration, and LU factorization. To broaden the scope beyond

purely numerical computation, this work also incorporates support for symbolic kernels in existing benchmarks as well as a naive Gröbner basis algorithm, enabling the evaluation of generic programming in workloads with irregular control flow, dynamic memory usage, and exact arithmetic.

A key aspect of the methodology is the use of uniform, minimally optimized implementations across all languages. Rather than leveraging highly tuned libraries or platform-specific optimizations, each algorithm is implemented in a straightforward manner to reduce the influence of platform-specific engineering decisions and better expose the interaction between language design, compiler behaviour, and runtime systems. This approach enables a controlled comparison of how different generic realization strategies behave under comparable conditions.

The contributions of this thesis are as follows:

- A cross-language benchmarking framework for evaluating the performance of generic and specialized implementations of scientific computing kernels.
- An extension of existing benchmark suites to include both numerical and symbolic workloads, enabling evaluation across a broader range of computational patterns.
- An empirical characterization of how different generic implementation strategies, including monomorphization, type erasure, and runtime specialization, influence the trade-offs between runtime performance, code size, and implementation flexibility.
- A synthesis of the trade-offs between abstraction and performance, relating these findings to different classes of applications in scientific and symbolic computing.

Through this analysis, the thesis provides a structured understanding of how abstraction mechanisms interact with language design and compiler strategy. Rather than presenting generics as uniformly efficient or costly, it suggests that their performance is strongly context-dependent, and it offers guidance on when generic abstractions can be expected to be effectively "free" and when they introduce significant overhead.

The remainder of this thesis is organized as follows. Chapter 2 reviews related work in scientific computing benchmarks, generic programming, and symbolic computation. Chapter 3 describes the overall approach, including language selection, algorithm selection, and benchmarking methodology. Chapter 4 details the implementation of the benchmark suite across the selected languages. Chapter 5 outlines the experimental platform and system configuration. Chapter 6 presents the results of the benchmarking experiments. Chapter 7

provides an analysis of these results, examining the performance characteristics observed across different languages and workloads. Finally, Chapter 8 concludes the thesis and discusses directions for future work.

Chapter 2

Related Work

Research on performance in scientific and symbolic computation spans several domains, including numerical benchmark suites, generic-programming frameworks, and computer algebra systems. This section reviews prior work relevant to this study, beginning with the paper this work derives from, followed by SciMark and its role as a numerical benchmark, continuing with the broader scientific-computing benchmark families, and concluding with computer algebra benchmarks that motivate the inclusion of a Gröbner basis kernel in this work.

2.1 Scientific Computing Benchmarks

Performance evaluation in scientific computing has traditionally relied on benchmark suites designed to capture the computational characteristics of numerical workloads. These benchmarks are typically composed of either full scientific applications or smaller computational kernels representing common algorithmic patterns such as linear algebra operations, spectral transforms, and iterative solvers. Their primary purpose is to evaluate the performance of hardware architectures, compilers, and runtime systems when executing workloads representative of scientific and engineering applications.

One widely used benchmark is the LINPACK benchmark, which measures system performance by solving dense systems of linear equations using LU decomposition[1]. LINPACK reports the sustained floating-point performance achieved during the computation and forms the basis of the TOP500 ranking of the world's fastest supercomputers. While

influential, LINPACK focuses on a single dense linear algebra workload with high computational intensity, and therefore does not capture the memory access patterns and algorithmic structures present in many scientific applications.

Other benchmark suites aim to represent a broader range of computational patterns. The NAS Parallel Benchmarks (NPB), originally developed by NASA, include several kernels and mini-applications derived from computational fluid dynamics workloads [2]. These benchmarks evaluate aspects of parallel system performance such as communication overhead, memory access behaviour, and scalability across distributed systems. Similarly, the HPC Challenge (HPCC) benchmark was designed to complement LINPACK by incorporating tests that stress different architectural components, including memory bandwidth, random memory access, and Fast Fourier Transforms [3].

While these benchmark suites primarily target the evaluation of hardware architectures and parallel systems, kernel-level benchmarks provide a more controlled environment for studying the performance characteristics of specific numerical algorithms. By isolating individual computational kernels, such benchmarks make it possible to analyze the effects of programming language features, compiler optimizations, and implementation strategies on numerical performance. This approach is particularly useful when the goal is to study how different programming abstractions influence the execution of common scientific computing operations.

2.2 Research Using Scientific Computing Benchmarks

Scientific computing benchmarks have frequently been used to evaluate the performance of programming languages, compilers, and runtime systems. Numerical kernels provide controlled workloads that expose the computational characteristics of scientific applications while remaining small enough for cross-language implementation.

Exploring and Benchmarking High Performance & Scientific Computing using R, R's HPC Packages and Lower level compiled languages : A Comparative Study

One example is the 2019 paper by Rahim K. Charania titled *Exploring and Benchmarking High Performance & Scientific Computing using R, R's HPC Packages and Lower level compiled languages : A Comparative Study.* In it, Charania explores the viability of using the R programming language for high performance and scientific computing by

benchmarking it against C/C++ while considering other advantages and drawbacks such as scalability and the usability of each language[4].

The paper begins by positioning the research around how R is used extensively for machine learning and neural networks due to its ease of use and the thousands of available libraries that require little coding. However, with increasing quantities of data being collected and more advanced algorithms being developed to analyze increasingly large datasets, interest in high performance computing for the language was growing.

The performance of R was then benchmarked against C/C++ through five different tests. These included a simple neural network classification task using the IRIS dataset, parallel averaging of one million random numbers using MPI and Rmpi, a scalability study of Rmpi across up to 20 cores, a Vector Auto-Regressive (VAR(1)) simulation implemented in both R and C++ via Rcpp, and parallel apply-style operations using the snow package.

Through these benchmarks, the author draws several conclusions. Firstly, C/C++ is dramatically faster for raw numerical computation which is expected because R is interpreted and dynamically typed while C++ code is compiled and optimized. Secondly, the trade-off for this reduced performance is improved expressiveness and developer productivity, as the author repeatedly observes that the equivalent R implementations are considerably shorter and require less implementation and maintenance effort. Thirdly, R's parallel packages (Rmpi, snow, snowfall) scale very well. This was most clearly supported by the Rmpi scalability study in which, although C is consistently much faster than R, as the number of cores increases for both implementations, the ratio of speed-up of C over R shrinks from a max of about 10 to a minimum of about 4.5 when increasing to 20 cores. Finally, the author finds that the C++ code produced from Rcpp can in fact be slower than pure R code due to the naive implementation from using Rcpp. The author argues that Rcpp only improves performance when the R code cannot be vectorized.

In summary, the paper concludes that while C/C++ remains superior for raw performance, R offers exceptional ease of use, rapid prototyping, and strong support for parallelism through packages like Rmpi, snow, and snowfall. Overall, R is presented as a highly productive environment for scientific computing, with parallel frameworks that scale well enough to offset many of its performance limitations.

This study follows a similar approach, evaluating scientific benchmarks across multiple languages. In contrast, explicit parallelism is avoided, and implementations between languages generally follow the same approach. Due to inherent differences in language capabilities and features, ease of use remains a dimension of study, though it is not the primary focus.

Performance Evaluation of Java against C and Fortran for Scientific Applications

Another example of research using scientific computing benchmarks is the 2001 study by Bull, Smith, Pottage, and Freeman titled *Performance Evaluation of Java against C and Fortran for Scientific Applications*. This work evaluates the viability of Java for scientific computing by comparing its performance against traditional high-performance languages such as C and Fortran[5].

The paper begins by discussing the benefits of Java for scientific applications, most notably its portability, as well as ease of use, benefits of just-in-time compilation, and strong networking support. However, these advantages comes with drawbacks in comparison to parallel programming models used in C and Fortran, such as MPI or OpenMP, at least in 2001. The authors also bring up the general user perception that Java had poor performance but that this study would help evaluate the validity of this perception.

The study utilizes the Java Grande Benchmark Suite, a collection of scientific workloads designed to represent common computational patterns found in numerical applications. These benchmarks include algorithms such as Fast Fourier Transforms, sparse matrix operations, and LU factorization, which are representative of the types of computations frequently encountered in scientific and engineering simulations.

The authors implemented these benchmarks across Java, C, and Fortran and executed them on several platforms using different compiler and virtual machine configurations. Their results showed that while Java generally performed slower than native languages, the performance gap was often smaller than expected due to improvements in just-in-time compilation and runtime optimization.

One key finding of the study was that Java performance varied significantly depending on the platform: systems based on Intel Pentium hardware running Linux exhibited relatively little performance loss, whereas the performance gap was much larger on Compaq Alpha systems. The authors argue that this reflects the efforts of each vendor in developing Java environments rather than intrinsic properties of the hardware.

This research demonstrates how scientific computing benchmarks can be used to evaluate the suitability of programming languages for computationally intensive applications. Similar to the study presented in this thesis, the authors compare multiple programming languages using representative scientific workloads. However, while their work focuses primarily on runtime performance differences between managed and native languages, the present study focuses on the performance implications of parametric polymorphism within scientific computing benchmarks.

2.3 SciMark

The most relevant scientific computing benchmark to this study is SciMark. SciMark 2.0 is a benchmark suite originally designed to evaluate the performance of numerical and scientific computing workloads on systems running the Java Virtual Machine (JVM) [6]. The benchmark measures the execution speed of several computational kernels and reports results in terms of approximate MFLOPS (millions of floating-point operations per second). The primary goal of the suite is to provide a simple and portable way to evaluate the floating-point performance of JVM implementations across different hardware and runtime environments.

Unlike large application benchmarks, SciMark focuses on a small set of well-understood numerical kernels that represent common patterns found in scientific computing. By measuring the execution time of these kernels, the benchmark provides an estimate of the floating-point performance of a system across several fundamental numerical operations.

SciMark provides two configurations: a standard version, which uses relatively small problem sizes that fit within the processor cache, and a LARGE version, which uses problem sizes that exceed cache capacity. This distinction allows the benchmark to evaluate both raw computational throughput and the performance implications of memory hierarchy and cache behaviour.

The benchmark suite consists of five primary kernels, each representing a different class of numerical computation frequently encountered in scientific workloads.

The Fast Fourier Transform kernel measures the performance of spectral transforms on complex data. FFT algorithms are widely used in signal processing, numerical analysis, and computational physics. The SciMark implementation uses a radix-2 Cooley-Tukey algorithm to transform a vector of complex values. This kernel stresses several aspects of a runtime environment, including complex arithmetic, non-sequential memory access patterns, and trigonometric operations. Because the algorithm repeatedly performs butterfly operations over large arrays, it is particularly sensitive to memory layout and the efficiency of arithmetic operations.

The Successive Over-Relaxation kernel models iterative grid-based numerical computations commonly found in the solution of partial differential equations. The algorithm repeatedly updates elements of a two-dimensional grid using a weighted average of neighbouring values. This kernel primarily evaluates memory access patterns and floating-point arithmetic within nested loops. Grid-based computations such as SOR are representative of many numerical simulations used in computational fluid dynamics, heat transfer, and other physical modelling applications.

The Monte Carlo kernel estimates the value of π by randomly sampling points within a unit square and determining how many fall inside the unit circle. While the mathematical computation itself is simple, the kernel exercises random number generation, conditional branching, and repeated function execution. Monte Carlo methods are widely used in scientific computing for numerical integration, probabilistic simulation, and uncertainty estimation. As a result, this kernel provides insight into how efficiently a runtime system handles repeated lightweight computations and random number generation.

Sparse matrix-vector multiplication represents computations where the majority of matrix elements are zero and therefore not explicitly stored. Such matrices commonly arise in scientific applications including graph algorithms, finite-element simulations, and large systems of linear equations. This kernel stresses irregular memory access patterns and indirect indexing, both of which are more difficult for compilers and hardware to optimize compared to dense numerical loops. The performance of this kernel therefore provides insight into how efficiently a system handles pointer-like indirection and non-uniform memory access.

The LU factorization kernel performs a matrix decomposition that expresses a matrix as the product of lower and upper triangular matrices. This operation forms the basis of many algorithms for solving systems of linear equations and is a fundamental routine in numerical linear algebra. Unlike the sparse matrix-vector kernel, LU factorization operates on dense matrices with regular memory access patterns. This allows the benchmark to measure the efficiency of tight numerical loops and floating-point arithmetic when operating on contiguous blocks of memory.

SciMark provides a useful benchmark suite for studying the performance implications of generic programming in scientific computing. The kernels represent a diverse set of computational patterns, including dense numerical loops, irregular memory access, stochastic simulation, and matrix operations. Together, they capture many of the fundamental behaviours of scientific workloads while remaining small enough to be easily ported across multiple programming languages.

Another important property of SciMark is that its implementations avoid heavy optimization or reliance on external numerical libraries. This allows the benchmark to isolate the behaviour of the programming language runtime and compiler rather than the performance of highly tuned native libraries. Because this study focuses on the performance impact of generic abstractions, using relatively straightforward algorithm implementations helps ensure that differences in performance are attributable primarily to language and runtime characteristics rather than algorithmic optimization.

Because SciMark focuses on relatively small numerical kernels with straightforward

implementations, it has been widely used in studies examining the performance of programming languages, virtual machines, and runtime systems for scientific workloads. Its simplicity makes it particularly suitable for controlled experiments in which the goal is to isolate the performance impact of specific language features or abstraction mechanisms. For these reasons, SciMark forms the numerical foundation of the benchmark suite used in this work. As in the study by Dragan and Watt, the SciMark kernels are implemented in both specialized and generic forms in order to measure the cost of abstraction introduced by generic programming.

However, SciMark focuses exclusively on floating-point numerical computation. To broaden the scope of the study, this work augments the benchmark suite with additional symbolic computation kernels, including a Gröbner basis algorithms. This extension allows the evaluation of generic programming performance not only in traditional numerical workloads but also in symbolic algebra applications, which exhibit very different computational characteristics.

2.4 Parametric Polymorphic Benchmarks

Parametric polymorphism, often referred to as generics, is a fundamental mechanism in modern programming languages that enables algorithms to be written independently of the specific types on which they operate. While this abstraction greatly improves code reuse and expressiveness, it can also introduce performance overhead depending on how the language implements polymorphism. As a result, several studies have investigated the performance implications of parametric polymorphism using benchmark workloads drawn from scientific computing.

A common approach in such studies is to implement algorithms in both specialized and generic forms and compare their performance across different language implementations. In languages such as C++, parametric polymorphism is typically implemented through templates, which are instantiated at compile time and often allow the compiler to generate highly optimized code often allowing performance comparable to specialized implementations. In contrast, languages such as Java and C# historically implemented generics using type erasure or runtime type abstractions, which can introduce additional indirection or runtime checks.

Several empirical studies have used numerical kernels to examine these differences. Scientific computing workloads are particularly well suited for this type of analysis because they consist of computationally intensive loops where even small abstraction overheads

can become visible in performance measurements. Kernels such as matrix factorization, spectral transforms, and iterative solvers provide controlled environments in which the costs of abstraction, dynamic dispatch, and memory indirection can be observed.

The work of Dragan and Watt represents one of the earliest systematic investigations of this topic in the context of scientific computing [7]. By implementing SciMark kernels using algebraic interfaces representing numeric structures, they demonstrated how different implementations of generics affect performance in numerical algorithms. Their results showed that while compile-time specialization can largely eliminate abstraction overhead, runtime-based generic implementations may introduce measurable costs depending on compiler optimization capabilities and runtime design.

Subsequent work in programming language performance evaluation has continued to explore these issues across evolving language ecosystems. Improvements in compiler technology, just-in-time compilation, and runtime specialization have reduced some of the overheads historically associated with generic abstractions. However, the interaction between generics, compiler optimization, and runtime behaviour remains complex and highly dependent on both the programming language and the workload being executed.

Because scientific kernels provide predictable computational structures and well-understood algorithmic behaviour, they remain an effective tool for evaluating the performance implications of parametric polymorphism. This work follows the same general methodology by implementing numerical and symbolic kernels in both specialized and generic forms, allowing the performance cost of abstraction to be measured directly.

2.5 Research using Parametric Polymorphic Benchmarks

A Comparative Study of Language Support for Generic Programming

An important study examining the design of generic programming systems is the 2003 paper by Garcia et al. titled *A Comparative Study of Language Support for Generic Programming*. The paper investigates how effectively different programming languages support the development of large-scale generic libraries and the language features required to enable flexible and reusable generic algorithms [8].

The paper begins by discussing the growing popularity and importance of generic programming, especially in spaces such as reusable libraries of software components, an example being C++'s Standard Template Library (STL). However, the authors argue that

it is important to understand what language features are necessary to support generic programming and to understand the extent to which specific languages support generic programming. Notably, the authors intentionally veer away from focusing on which language is "better" than others.

To conduct this analysis, the authors implement a generic graph library across several programming languages, including C++, Standard ML, Haskell, Eiffel, Java (using a proposed generics extension), and Generic C#. The design of the library is inspired by the Boost Graph Library and includes a number of classical graph algorithms such as breadth-first search, Dijkstra's shortest path algorithm, Prim's minimum spanning tree algorithm, and Johnson's all-pairs shortest paths algorithm. Rather than focusing on runtime performance, the implementations are used to evaluate how well each language expresses generic abstractions and supports the requirements of concept-based generic programming.

Through this comparative implementation, the authors identify several key language features that significantly impact the usability and expressiveness of generic programming systems. These include the ability to express constraints on type parameters, support for associated types, retroactive modelling of interfaces, separate compilation, implicit instantiation of templates, and concise syntax for specifying generic requirements. The study finds that languages lacking these features often require workarounds that lead to more verbose code, reduced modularity, and less maintainable implementations. While C++ provides a powerful template system capable of expressing highly generic algorithms, the authors note that the lack of explicit concept checking can lead to complicated and difficult-to-diagnose compilation errors.

While Garcia et al. focus primarily on evaluating the language design and expressiveness of generic programming systems, their work does not directly measure the runtime performance implications of generic abstractions. In contrast, the present study focuses on the performance characteristics of generic implementations within scientific computing benchmarks. By evaluating benchmark kernels implemented using parametric polymorphism across multiple programming languages, this work complements the findings of Garcia et al. by examining the practical performance costs associated with generic abstractions in computational workloads.

Generic Go to Go: Dictionary-Passing, Monomorphisation, and Hybrid

The introduction of generics in Go 1.18 motivated research into how parametric polymorphism should be implemented efficiently in the language. One such study is the 2022 paper by Ellis et al. titled *Generic Go to Go: Dictionary-Passing, Monomorphisation, and Hy-*

brid. The paper investigates compilation strategies for implementing generics in Go and proposes a new translation approach designed to improve upon existing implementations [9].

The authors begin by examining existing approaches to compiling generics, including monomorphisation, dictionary-passing, and hybrid approaches that combine elements of both. Go 1.18 adopts a hybrid strategy in which specialized implementations are generated through a form of monomorphisation while dictionaries are used to recover type information lost during specialization. While this approach can achieve strong runtime performance, it may also lead to increased code size, slower compilation times, and limitations in the range of programs that can be translated.

To address these issues, the authors propose a new non-specializing call-site dictionary-passing translation from Featherweight Generic Go (FGG) to Featherweight Go (FG). In this approach, dictionaries representing type information are generated at the call site of generic functions rather than through global specialization. The translation is formally defined and proven correct using a bisimulation-based proof technique that establishes behavioural equivalence between the original generic program and the translated program.

To evaluate the practical implications of different compilation strategies, the authors implement and compare five translators: the Go 1.18 implementation, the Go team's prototype translator (`go2go`), a monomorphisation translator, an erasure-based translator, and their proposed dictionary-passing translator. These implementations are evaluated using both microbenchmarks and reimplemented real-world generic programs, measuring execution time, compiled code size, and compilation time.

The results highlight clear trade-offs between the different compilation strategies. Monomorphisation tends to produce the fastest executable programs but significantly increases code size and compilation time due to the generation of specialized code for each type instantiation. Dictionary-passing approaches produce smaller binaries and faster compilation times but may incur modest runtime overhead due to additional indirection. The authors demonstrate that their call-site dictionary-passing translation achieves competitive runtime performance while reducing some of the code size and compilation overhead associated with monomorphisation.

This work relates to the present study through its investigation of the performance implications of parametric polymorphism. However, while Ellis et al. focus primarily on the implementation and compilation strategies used to support generics within a single language, the present research evaluates the runtime performance of generic implementations across multiple programming languages using standardized scientific computing benchmarks. As such, this work complements the present study by examining how generics are

implemented at the compiler level, whereas the present research evaluates the observable performance impact of generic abstractions within computational workloads.

2.6 SciGMark

This paper takes inspiration from and builds directly upon the 2005 study by Laurentiu Dragan and Stephen Watt titled *Performance Analysis of Generics in Scientific Computing*. Their work investigated the performance implications of generic programming in scientific computing applications across several programming languages[7].

Scientific computing workloads often rely on highly optimized numerical kernels, where even small inefficiencies can have significant performance consequences. At the same time, generic programming techniques allow developers to write algorithms abstractly so that they can operate on multiple numeric types without code duplication. Dragan and Watt examined the tension between these two goals: the desire for abstraction and the need for high-performance numerical computation.

Their study evaluated the performance of generics in three programming languages: C++, C#, and Java. These languages implement generics using different mechanisms, including template-based compile-time specialization and runtime generic abstractions. As a result, they provide a useful basis for comparing how language design choices influence the cost of abstraction in numerical software.

To conduct their analysis, Dragan and Watt implemented both specialized and generic versions of several numerical kernels drawn from the SciMark benchmark suite to create the benchmark suite that they call SciGMark. These kernels represent common workloads in scientific computing and include: Fast Fourier Transform (FFT), Successive Over-Relaxation (SOR), Monte Carlo integration for approximating π, Sparse matrix-vector multiplication, and Dense LU matrix factorization. In addition to the SciMark kernels, the authors introduced a polynomial multiplication benchmark, extending the benchmark suite beyond purely numerical operations and allowing them to evaluate generics in an algebraic computation context.

A key component of their methodology involved designing a set of interfaces representing algebraic structures. Rather than implementing algorithms directly for a specific numeric type such as floating-point numbers, they defined interfaces describing the operations available on a class of values. For example, one interface defined the operations of a field, including addition, subtraction, multiplication, and division, while another interface described operations such as absolute value and square root. Numerical algorithms were

then implemented generically using these interfaces, allowing the same algorithmic code to operate on multiple underlying types. This approach enabled the authors to compare two implementation strategies: specialized implementations where algorithms are written directly for a concrete numeric type, and generic implementations where algorithms operate on abstract interfaces representing numeric capabilities. By benchmarking both versions of each kernel, they were able to measure the performance cost introduced by generic abstraction.

Their results showed that while generic programming provides significant advantages in terms of code reuse and expressiveness, it can introduce non-trivial performance overheads in performance-critical applications. The magnitude of this overhead varied substantially between languages and depended heavily on the mechanisms used to implement generics. In particular, differences in compiler optimization strategies and runtime dispatch mechanisms played an important role in determining the final performance characteristics.

The authors concluded that the interaction between generics, compiler optimizations, and runtime systems remains complex and not fully understood. Their work highlighted the need for further empirical study of generic programming in scientific workloads, particularly as programming languages continue to evolve and adopt new generic programming models.

While the study provided important early insights into the performance implications of generic abstractions, it was limited in scope. The set of programming languages considered was restricted to those widely used in 2005, and the benchmark suite focused primarily on numerical kernels derived from SciMark. Since then, both programming languages and generic programming techniques have evolved significantly.

The present work builds upon the framework established by Dragan and Watt by extending their approach to additional languages and workloads. In particular, this study expands the benchmark suite to include symbolic computation tasks, such as Gröbner basis calculations, in addition to the numerical kernels used in the original study. By doing so, it aims to provide a broader perspective on the performance implications of generic programming in both numerical and symbolic scientific computing.

2.7 Computer Algebra Benchmarks

While scientific computing benchmarks focus on numerical workloads such as floating-point arithmetic, a separate class of benchmarks exists solely for evaluating symbolic computation. Computer algebra benchmarks measure the performance of systems on tasks involving

algebraic structures rather than numerical values. These workloads typically involve operations such as polynomial arithmetic, term rewriting, expression simplification, and the manipulation of symbolic objects. As a result, they evaluate different aspects of a language runtime, including dynamic memory allocation, recursion, irregular control flow, and the overhead of generic abstraction.

Symbolic benchmarks are commonly used in the evaluation of computer algebra systems such as Maple and Mathematica. An essential component of these benchmarks is polynomial ideal computation, particularly Gröbner basis algorithms. Standard benchmarks include the cyclic and Katsura polynomial systems. These benchmarks were designed to expose the combinatorial complexity inherent to symbolic algebra, where intermediate expression growth and non-uniform computation patterns dominate performance.

Compared to numerical kernels, symbolic workloads are far less regular as they generally lack the consistent, tight loops over arrays characteristic of numerical kernels, and they rarely benefit from hardware-level optimizations such as vectorization or cache blocking. Instead, they emphasize the cost of abstraction, the efficiency of data structures, and the responsiveness of the runtime to dynamically evolving computations. For this reason, symbolic benchmarks complement numerical ones by highlighting performance characteristics that would otherwise remain hidden.

In this work, symbolic algebra kernels are included, most notably a naive Gröbner basis implementation. Although simplified relative to state-of-the-art algorithms, this naive implementation captures the essential features of symbolic workloads: polynomial reduction, term rewriting, and irregular control flow. Its inclusion expands the scope beyond SciMark's purely numerical focus and enables the evaluation of generic programming performance on non-numeric, structurally complex computations. This provides a broader and more balanced perspective on how different languages and runtimes handle generic abstractions across both numerical and symbolic domains.

2.8 Research using Computer Algebra Benchmarks

Comparing the Speed of Programs for Sparse Polynomial Multiplication

The 2003 work of Richard Fateman investigates the performance of sparse polynomial multiplication across a range of computer algebra systems (CAS). Using a benchmark based on expanding and multiplying multivariate polynomials such as $(1+x+y+z)^{20}$, the study evaluates how different implementation strategies affect execution time and resource usage [10].

This work is situated within the domain of symbolic computation and CAS performance evaluation. Rather than proposing new algorithms, it focuses on empirical comparison of existing approaches, positioning itself as a practical benchmark study. It complements theoretical work on algebraic algorithms by emphasizing real-world system behaviour, particularly in the presence of large symbolic expressions and arbitrary-precision arithmetic.

The study defines a benchmark problem involving sparse multivariate polynomial multiplication and evaluates multiple implementations across different systems and representations. Several data structures are considered, including recursive lists, vectors, and hash tables, each with different memory and locality characteristics. The implementations are tested under consistent conditions, with attention paid to factors such as coefficient arithmetic (including use of GMP), memory allocation, and cache behaviour. Performance is measured primarily in terms of execution time, with additional analysis of garbage collection and cache efficiency.

The results demonstrate that data representation and memory behaviour dominate performance, often outweighing algorithmic complexity. In particular, dense representations using contiguous memory (e.g., vectors) outperform pointer-based structures due to improved cache locality. The study also finds that asymptotically faster algorithms such as Karatsuba multiplication and FFT-based methods are not well-suited to sparse inputs, where they introduce unnecessary overhead. Additionally, the cost of arbitrary precision arithmetic and garbage collection is shown to significantly impact runtime. Overall, the work concludes that efficient memory access patterns and careful implementation choices are critical for high performance symbolic computation.

This research directly informs the present study's focus on benchmarking scientific and symbolic computation across programming languages. While the benchmark problems differ (polynomial multiplication versus linear algebra and numerical kernels), both studies highlight the importance of selecting representative workloads and interpreting results in the context of system-level behaviour rather than purely theoretical complexity.

Relax, but Don't Be Too Lazy

The 2002 paper *Relax, but Don't Be Too Lazy* examines the impact of evaluation strategies, particularly the balance between eager and lazy computation, on the efficiency of symbolic and algebraic systems. It explores how deferring computation can both improve and degrade performance depending on the context [11].

This work lies at the intersection of programming language design and symbolic computation. It contributes to a broader body of research on evaluation strategies and their

performance implications, particularly in languages and systems that support symbolic manipulation. Unlike purely systems-oriented benchmarks, this work focuses on semantic and runtime design choices, offering insight into how language features influence computational efficiency.

The paper analyzes different evaluation strategies by examining how expressions are computed under varying degrees of laziness. The work does not evaluate specific programming languages empirically, but instead analyzes evaluation strategies-such as lazy and eager computation-that underpin many modern programming languages, including Haskell and ML-family languages. It considers scenarios in which computations are deferred versus executed eagerly, and studies the resulting trade-offs in terms of overhead, redundancy, and optimization opportunities. The approach is primarily analytical, supported by illustrative examples and performance considerations within symbolic computation contexts.

The key conclusion is that while laziness can avoid unnecessary computation, excessive deferral introduces overhead and can hinder optimization, ultimately reducing performance. In particular, overly lazy evaluation may lead to increased bookkeeping, delayed simplification, and missed opportunities for efficient execution. The paper argues for a balanced approach, where evaluation strategies are chosen carefully to match the computational workload and system design.

This work is relevant to the present study's investigation of programming language performance in scientific and symbolic computation. It highlights how language-level design decisions, such as evaluation strategy, can significantly affect runtime behaviour, independent of algorithmic complexity. This is particularly important when comparing languages with different execution models, as differences in evaluation strategy may influence the efficiency of benchmarks. As such, this paper provides conceptual grounding for interpreting performance differences observed in this thesis, especially in cases where abstraction and execution semantics play a role.

Chapter 3

Approach

From a high-level perspective, this project evaluates how different platforms implement and execute generic algorithms by implementing a shared suite of computational kernels across multiple platforms. The guiding principle was to minimize platform-specific optimizations and instead adopt a uniform, naïve implementation strategy. This helps ensure that the comparison focuses on each platform's generic programming model rather than any ecosystem of micro-optimizations or specialized libraries. By constraining the implementations to a common baseline, the study seeks to reduce the influence of platform-specific optimizations and better expose the performance trade-offs associated with different generic programming models.

3.1 Specialized vs Generic

A natural way to evaluate a platform's support for generics is to compare two versions of the same algorithm: a specialized implementation using built-in primitive types, and a generic implementation parameterized over abstract interfaces.

Specifically, specialized implementations operate on concrete data types, where functions operate on fixed concrete representations of data. In the context of scientific computing, this typically includes primitive numeric types such as floating-point numbers or integers, as well as domain-specific representations such as finite fields. These implementations benefit from direct hardware support, compiler optimizations, and minimal indirection and avoidance of dynamic dispatch[12].

In contrast, generic implementations operate on abstract interfaces. This allows algorithms to be expressed independently of the underlying data type, enabling reuse across multiple domains. For example, a single implementation may operate on floating-point numbers, complex numbers, finite fields, or exact arithmetic types such as rational or modular integers, provided that the required algebraic operations are defined. Nevertheless, this abstraction can introduce several drawbacks, including increased implementation complexity and potential performance overhead. Such overhead may arise from additional layers of indirection, reduced opportunities for compiler specialization, dynamic dispatch, interface lookup, and increased memory footprints[12].

To illustrate this distinction, consider the Java implementations used in this study. The specialized version of the Fast Fourier Transform (FFT) operates directly on the *double* primitive type. In contrast, the generic version uses a custom *DoubleField* class, which encapsulates a *double* value while implementing the operations required of a field (e.g., addition, subtraction, multiplication, and division). The Java Virtual Machine (JVM) can apply substantial optimizations to computations involving primitive *double* values during just-in-time (JIT) compilation. However, the use of *DoubleField* introduces object allocation, method dispatch, and additional indirection, which may reduce opportunities for certain compiler optimizations and introduce additional runtime overhead.

It is also important to note that platforms differ substantially in how generic abstractions are implemented and compiled. For example, Go uses implicit interface satisfaction, meaning that types do not explicitly declare which interfaces they implement. As a result, compatibility with a generic algorithm is only validated when the type is used within the relevant context.

More broadly, parametric polymorphism may be implemented using either homogeneous or heterogeneous strategies. In homogeneous systems, type information is typically erased and operations are performed over a uniform runtime representation, often requiring indirection or casting. Heterogeneous systems instead generate specialized code for concrete type instantiations. These implementation strategies influence both runtime behaviour and code generation characteristics.

These aspects are discussed in further detail in subsequent sections, where each platform's approach to generics is examined in the context of the benchmark implementations.

3.2 Language Selection

The selection of languages is critical to ensuring that the study captures a representative range of generic programming and compilation strategies. Several dimensions of the design space are considered, as well as popularity and practical usage of each language. Some of these dimensions include how memory is managed, what compilation strategy is used, what type system is used, and how generic abstractions are realized.

The primary benchmark suite focuses on Rust, Java, Go, and TypeScript. This set was selected to represent a broad range of language capabilities while also reflecting language popularity. Several languages were also considered but ultimately not selected for the full test suite, either due to time constraints or overlap with other languages. However, to provide some insight into the languages, both Julia and C++ also have include implementations of the specialized and generic naive Gröbner basis benchmark.

3.2.1 Rust

Rust represents the modern systems-programming paradigm, designed to provide memory safety and high performance without relying on a garbage collector. Its generic system is based on monomorphization, meaning that for every concrete type used with a generic function or data structure, the compiler generates a specialized version of the code. This approach is often associated with the goal of "zero-cost abstractions", where generic code can approach the performance of hand-written specialized implementations when compiler optimizations are successful[13].

Rust's monomorphization strategy has several implications for this study. First, it provides a best-case scenario for generic performance: generic algorithms can incur little to no runtime penalty relative to their specialized counterparts. Second, because Rust performs ahead-of-time (AOT) compilation using LLVM, the compiler has extensive visibility into the concrete types and can aggressively optimize across function boundaries. This includes inlining, constant propagation, and vectorization, all of which benefit numeric kernels such as FFT and matrix operations[14].

Rust's nominal type system and trait-based abstraction model also play a role. Traits allow developers to express algebraic structures (e.g. fields) in a way that is both type-safe and amenable to optimization[15]. However, Rust's strict ownership and borrowing rules introduce additional complexity when implementing algorithms that require frequent cloning or temporary values, especially in symbolic computing contexts. This tension between safety and ergonomics is an important part of Rust's profile in this study.

Overall, Rust serves as a reference point for what is achievable when generics are compiled heterogeneously and optimized ahead of time. Its results help contextualize the performance costs observed in languages that rely on type erasure or runtime specialization.

3.2.2 Java

Java provides a contrasting model built around homogeneous generics implemented via type erasure. At compile time, generic type parameters are removed, and all generic operations are performed on erased types (typically *Object*) with casts inserted where necessary[16]. This design preserved backward compatibility with older JVM bytecode but limits the ability of the compiler and JIT to specialize generic code for specific types[17].

Java's runtime environment is built around a garbage-collected heap and a sophisticated just-in-time compiler. The HotSpot JVM can optimize hot loops aggressively, including inlining, escape analysis, and speculative optimizations. At the same time, because type information is erased, the JVM does not perform systematic ahead-of-time monomorphization of erased generic types. As a result, generic implementations often incur additional overhead from object allocation, virtual method dispatch, and the inability to use primitive types directly.

The distinction is particularly important for scientific computing workloads. Specialized Java implementations using the *double* primitive benefit from hardware-level floating-point operations and JIT optimizations. In contrast, generic implementations must wrap numeric values in objects, which increases memory usage and introduces indirection. These differences make Java a useful case study for understanding the performance penalties associated with homogeneous generics.

Java's nominal type system and mature ecosystem also influence implementation choices. While interfaces provide a clean way to express algebraic structures, the lack of value types means that generic numeric abstractions typically incur object overhead. Although Project Valhalla aims to introduce value types in future JVM releases, these features are not yet available in mainstream Java and, therefore, are not under consideration for this study. This makes Java a representative example of the trade-offs inherent in type-erased generic systems.

3.2.3 Go

Go occupies a middle ground between Rust and Java in both language philosophy and generic implementation strategy. Introduced in Go 1.18 (March 2022), generics in Go com-

bine shape-based compilation techniques with dictionary passing mechanisms, depending on the form of the generic code and constraints involved. This hybrid approach aims to balance performance with binary size and compilation speed[18].

Go's type system is structural and interface-based, meaning that types satisfy interfaces implicitly. This design makes generic programming flexible but also introduces challenges: whether a type satisfies an interface is not explicit, and errors may surface only when generic functions are instantiated. For this study, this means that generic numeric abstractions must be carefully designed to ensure that all required operations are available and correctly implemented.

Go uses a garbage collector and ahead-of-time compilation. Its GC is optimized for low-latency server workloads rather than high-throughput numeric computation, which can influence performance for algorithms that allocate many small objects, such as generic numeric wrappers[19].

Additionally, the Go FAQ states that one of the language's design goals is extremely fast compilation, with the goal that even large executables should build in only a few seconds[19]. This emphasis on compilation speed may reduce the extent of aggressive optimization relative to systems such as LLVM or JIT systems like HotSpot.

Despite these limitations, Go's generics represent an important point in the design space. The hybrid compilation strategy allows some specializations while still supporting interface-based abstraction. This makes Go a valuable case study for understanding how modern languages attempt to incorporate generics without sacrificing simplicity or compilation speed.

3.2.4 TypeScript

TypeScript occupies the dynamically interpreted end of the spectrum, despite providing a static type system that is erased before execution. Its generics exist purely at compile time and are erased during the transpilation to JavaScript. At runtime, type information from TypeScript is no longer available, and generic abstractions have no direct representation. Consequently, TypeScript can be viewed as an example of a purely homogeneous generic system, where all type information is removed before execution[20].

Because TypeScript compiles to JavaScript, its performance characteristics are determined by the JavaScript engine. Modern JavaScript engines such as V8 rely on just-in-time compilation, hidden classes, and inline caches to optimize property access and hot code paths. As documented in V8's "Fast Properties"[21] and "Hidden Classes"[22] articles,

these optimizations depend on stable object shapes: changes to a hidden class, transitions to dictionary-mode properties, or polymorphic access patterns can prevent the optimizing compiler and inline caches from generating optimal code. As a result, numeric abstractions implemented as objects or classes-such as generic numeric wrappers-can introduce shape changes or dictionary properties that degrade the performance of hot loops.

TypeScript's structural type system also influences how algebraic abstractions are expressed. Interfaces exist only at compile time, so runtime checks must be implemented manually if needed. This limits the ability to enforce algebraic constraints and increases the likelihood of runtime errors if generic functions are misused.

Memory management is fully dynamic, with garbage collection handled by the JavaScript runtime. This introduces overhead for object-heavy generic implementations, especially when compared to specialized numeric arrays. As a result, TypeScript provides a useful contrast to the other languages: it demonstrates how generic abstractions behave in a dynamic environment where type information is not available to the compiler or runtime.

3.2.5 Julia

Julia represents a high-level, high-performance language specifically designed for scientific and numerical computing. Unlike the other languages in this study, Julia was built with generic programming and multiple dispatch as core design principles. Its type system is dynamic but strongly typed, and its generics are implemented through parametric methods combined with just-in-time (JIT) specialization.

A key feature of Julia is that generic functions are compiled using type specialization at runtime. When a function is first invoked with a particular set of argument types, the Julia compiler generates a specialized version of that function for those types. This behaviour is resembles monomorphic specialization, as each concrete instantiation results in optimized machine code. In practice, this can allow type-stable Julia code to approach the performance of statically compiled languages while retaining the flexibility of a dynamic language[23].

Julia's use of multiple dispatch further distinguishes it from the other languages considered. Rather than associating methods with a single receiver type, Julia selects implementations based on the combination of all argument types. This model is particularly well-suited for expressing algebraic structures and operations, as it allows concise and extensible definitions of arithmetic over different domains (e.g., integers, floating-point numbers, finite fields, or polynomials). For symbolic computation tasks such as Gröbner basis algorithms, this flexibility can significantly simplify implementation[24].

The runtime environment of Julia includes a garbage collector and a JIT compiler built on LLVM. The compiler performs substantial specialization and optimizations once type information is known, including inlining and specialization across function boundaries. At the same time, because compilation occurs at runtime, Julia introduces latency during the first execution of a function (often referred to as "time to first plot"). This overhead can be significant for workloads involving many distinct type instantiations or short-lived computations[25].

From the perspective of this study, Julia represents a language specifically designed around generic scientific computing: it combines high-level abstractions with type-driven specialization. However, its reliance on JIT compilation and runtime type inference introduces trade-offs that differ from both fully ahead-of-time compiled systems like Rust and C++, and purely dynamic environments like TypeScript. As such, Julia provides valuable insight into how dynamic languages can achieve high performance through specialization.

3.2.6 C++

C++ represents the traditional systems-programming approach to high-performance generic programming. Its template system provides one of the earliest and most influential implementations of heterogeneous generics, where templates are instantiated at compile time to produce specialized code for each set of template arguments. This mechanism is directly analogous to Rust's monomorphization and is widely associated with the goal of "zero-cost abstractions" in performance-critical code[26].

Templates in C++ are purely compile-time constructs, allowing the compiler to have extensive visibility into types and operations. This enables aggressive optimization, including inlining, constant folding, and loop transformations, particularly when combined with modern optimizing compilers such as Clang and GCC. As a result, C++ has long been the dominant language for scientific computing libraries, including linear algebra frameworks and computer algebra systems.

Memory management in C++ is manual by default, though modern practices encourage the use of RAII (Resource Acquisition Is Initialization) and smart pointers to manage lifetimes safely. The absence of garbage collection avoids garbage-collector-related runtime overhead, which can be advantageous for allocation-heavy workloads. However, it also places greater responsibility on the developer to ensure correctness, particularly in complex symbolic algorithms[27].

In this study, C++ serves as a baseline for high-performance generic programming, particularly for the Gröbner basis implementation. Its mature ecosystem and highly opti-

mized compilation model provide a point of comparison for newer languages such as Rust, as well as for runtime-specializing systems like Julia.

3.3 Algorithm Selection

As discussed in Chapter 2, this work extends prior benchmark studies, notably SciMark, and therefore retains its core set of algorithms for consistency. To broaden the scope, we also add support for finite fields to provide representation for symbolic computing tests, most notably, this includes a naive Gröbner basis algorithm.

This project extends the earlier benchmark implementations to support a broader range of data types. For example, while SciMark originally used the *double* primitive, Dragan and Watt later generalized the implementation through the introduction of the *DoubleRing* abstraction that replaced the *double* primitive while also introducing a *Complex* generic class for use in FFT. Building on this, the present work introduces finite-field arithmetic via the *IntModP* class. Supporting such algebraic structures required modifications to the FFT implementation, particularly to enable the computation of roots of unity across different domains. This extension illustrates the expressive power and flexibility afforded by generics.

The naive Gröbner basis algorithm is selected as a complementary benchmark due to its computational characteristics. Despite its relatively small memory footprint compared to algorithms such as FFT, it exhibits high computational intensity. This is evident in the time required to compute bases for instances of the cyclic n-roots problem. Unlike the other benchmarks, which typically perform a single pass over the input data, the Gröbner basis algorithm is inherently recursive and repeatedly transforms the input basis until a fixed point is reached. Another motivation for selecting the naive Gröbner basis is that it is a symbolic computing algorithm, which brings with it a need for exact arithmetic. This introduces new challenges for the platforms that were not present in the numerical benchmarks, highlighting limitations in platforms that rely heavily on floating-point operations. Consequently, support for alternative algebraic structures, such as finite fields, becomes necessary, as floating-point representations are insufficient for exact computation.

Other candidate benchmarks, such as those from the Stanford benchmark suite, were considered. However, many were excluded due to their limited emphasis on mathematically intensive operations. The selected algorithms were chosen specifically to highlight differences in generic realization strategies and runtime behaviour.

3.4 Benchmarking

3.4.1 Performance Measurement

A central challenge in this study is establishing a fair basis for performance comparison across platforms with fundamentally different execution and compilation models. To address this, a conservative and uniform measurement strategy is adopted: all execution times are obtained using the Unix *time* command. This approach reduces variability introduced by platform-specific timing mechanisms and reduces some potential sources of measurement inconsistency.

This choice, however, limits the ability to fully account for runtime-specific optimizations, particularly in languages that employ just-in-time (JIT) compilation. Since such systems require warm-up periods to reach peak performance, direct comparisons across languages must be interpreted with care. Nevertheless, the primary focus of this study is the relative performance difference between specialized and generic implementations within each platform, rather than absolute cross-language comparisons.

To partially mitigate the effects of JIT compilation, algorithms that are not inherently iterative (i.e., those excluding Monte Carlo and SOR) are executed in a loop of ten iterations within a single run. This allows JIT-based systems to optimize during early iterations while ensuring that the measured time is more representative of steady-state behaviour. Additionally, this approach amortizes setup costs such as input generation and object initialization.

Input sizes are determined by the constraints of the platform with the highest memory overhead. For instance, Java's object-heavy generic implementations may exhaust heap space at smaller input sizes compared to Rust's monomorphized equivalents. Accordingly, the largest input size is selected as the maximum size that can be executed reliably across all platforms. Two smaller input sizes are then chosen to provide a range of execution times.

For Gröbner basis computations, input selection follows a different strategy due to the recursive and input-sensitive nature of the algorithm. Specifically, instances from the cyclic n-roots family are used, with $n = 4, 5$, and 6, the latter representing the practical upper bound under the given resource constraints.

3.4.2 Output Artifact Sizes

In addition to runtime performance, this study considers the size of compiled artifacts as a secondary metric, providing insight into trade-offs associated with different compilation strategies. Three primary factors influence artifact size: compilation target, linking strategy, and runtime architecture.

The compilation target refers to the form of the generated output, whether native machine code, intermediate representation, or source-level code. The linking strategy determines the extent to which external libraries are incorporated into the final artifact, distinguishing between static linking (where dependencies are embedded) and dynamic linking (where dependencies are resolved at runtime). Finally, the runtime architecture defines whether execution relies solely on the operating system or requires an additional runtime environment, which may either be bundled with the artifact or provided externally.

Given the diversity of approaches across platforms, a tailored methodology is adopted for each to ensure meaningful comparison.

Rust

Rust employs an ahead-of-time (AOT) compilation model with statically linked binaries and relies on the operating system for runtime support. Consequently, artifact size is measured directly as the size of the compiled executable.

Due to Rust's monomorphization strategy, generic implementations are expected to produce larger binaries than their specialized counterparts. However, static linking introduces a substantial baseline size due to included libraries, which may obscure these differences. To avoid inflating measurements, the total footprint of the full benchmark suite is obtained by compiling a single executable containing all tests, rather than summing individual binaries.

Java

Java represents the opposite end of the spectrum, utilizing an intermediate representation (bytecode), dynamic linking, and an external runtime in the form of the JVM. As a result, individual artifacts are significantly smaller, with much of the functionality delegated to the runtime environment.

To enable fair comparison, artifact size measurements include all relevant class files associated with a given benchmark. For example, the footprint of the generic Gröbner

basis implementation includes not only the main class but also supporting classes (e.g., *IntModP*, *VecExponent*) and interfaces (e.g., *IField*, *IMath*).

Go

Go occupies an intermediate position, combining ahead-of-time compilation with statically linked binaries and an embedded runtime. This design emphasizes deployment simplicity and self-contained executables, though at the cost of increased binary size.

As with Rust, the total footprint of the full test suite is measured using a single compiled executable containing all benchmarks, while individual tests are measured separately when needed.

TypeScript

TypeScript differs fundamentally from the previous languages in that it is transpiled to JavaScript rather than compiled to machine code. The resulting JavaScript files are dynamically linked and executed via an external runtime (Node.js), which employs JIT compilation.

Accordingly, artifact size is defined as the combined size of all generated JavaScript files relevant to each benchmark. For specialized implementations, this typically corresponds to a single file, whereas generic implementations may involve multiple modules.

C++

C++ supports both static and dynamic linking, although, in this study, the default configuration (dynamic linking) is used. Because of this, the resulting executable depends on the presence of standard C++ runtime libraries. Additionally, as an AOT-compiled language that relies on the operating system for runtime support, it typically produces relatively small binaries.

Artifact size is therefore measured as the size of the compiled executable. A minor caveat is that the generic Gröbner basis implementation in C++ uses slightly fewer abstractions than in other platforms, which may marginally affect comparability.

Julia

Julia, like TypeScript, distributes source code rather than standalone binaries, but differs in that it executes code directly using its own runtime (Julia) and JIT compiler. Unlike transpiled systems such as TypeScript, Julia source files are compiled dynamically at runtime by the Julia compiler and JIT infrastructure.

Thus, artifact size is measured as the total size of all relevant `.jl` source files associated with each benchmark.

Chapter 4

Implementation

The primary goal was to maintain broadly comparable, language-agnostic implementations in order to provide a more consistent basis for evaluating generic and specialized implementations across platforms. Although the emphasis of the study is primarily on intra-language comparisons between specialized and generic implementations, implementations were intentionally designed to avoid highly platform-specific optimizations where possible. Instead, the benchmarks follow relatively straightforward and minimally optimized designs intended to better expose differences in language design, compiler behaviour, runtime representation, and generic realization strategy. Some language-specific conveniences are still used where necessary, though these are generally not expected to substantially affect performance characteristics; cases where they may do so are discussed further in Chapter 7.

4.1 Algorithm Overview

4.1.1 Monte Carlo Integration

Monte Carlo integration estimates areas by sampling random points and checking what fraction fall inside a region. To estimate π, this implementation embedded a quarter-circle of radius 1 inside the unit square $[0, 1] \times [0, 1]$. The area of this quarter-circle is $\pi/4$. As such, if points within the unit square are sampled uniformly at random, the probability that a point lands inside the quarter-circle is also $\pi/4$. This provides the estimator $\pi \approx 4 \times$ (number of points in the quarter-circle $\div$ number of total sampled points). Therefore, with enough iterations and sufficient random selection, the estimator will converge to π.

4.1.2 Successive Over-Relaxation

Successive Over-Relaxation (SOR) applies a weighted 5-point averaging stencil over a 2D grid, using updated neighbour values and a relaxation factor ω to accelerate convergence toward a steady-state configuration. In this benchmark, SOR is used purely as an iterative relaxation kernel rather than as a linear-system solver. Each sweep updates the grid in place, producing the characteristic memory-bound access pattern found in finite-difference diffusion methods such as 2D Laplace relaxation with Dirichlet boundaries. By varying the grid size while keeping the iteration count fixed, the benchmark probes different working-set regimes, providing insight into how different runtimes handle stencil-dominated workloads.

4.1.3 LU factorization

LU factorization computes a dense in-place decomposition of an $n \times n$ matrix using classical Gaussian elimination with partial pivoting. For each column, the algorithm selects the pivot as the entry of largest absolute value, swaps rows to bring that pivot to the diagonal, and then performs the standard elimination step by storing the multipliers below the diagonal and applying a rank-1 update to the trailing submatrix. The result overwrites the input matrix with the combined L and U factors, while the sequence of row swaps encodes the permutation. This kernel is dominated by structured floating-point arithmetic and contiguous memory access, with pivoting adding predictable row-movement overhead. As a benchmark, dense LU stresses arithmetic throughput, cache behaviour, and branch-predictable control flow typical of dense linear-algebra workloads.

4.1.4 Fast Fourier Transform

The Fast Fourier Transform (FFT) kernel performs a one-dimensional forward transform of a fixed-size vector using the iterative radix-2 Cooley-Tukey algorithm over the coefficient domain. The computation begins with an in-place bit-reversal permutation and proceeds through a sequence of butterfly stages, each combining pairs of values with twiddle factors derived from a suitable principal root of unity. This creates a workload that mixes structured arithmetic with non-contiguous memory accesses and stride-dependent data shuffling, making FFT a representative benchmark for irregular access patterns and array-indexing overhead across languages.

4.1.5 Gröbner Basis

The Gröbner basis kernel implements the naive form of Buchberger's algorithm for multivariate polynomial ideals. The algorithm iterates over all polynomial pairs, constructs their S-polynomials, and performs full multivariate polynomial division to reduce each S-polynomial with respect to the current basis. If the remainder is nonzero, it is added to the basis and the process continues until no new polynomials are added. This implementation omits many common optimizations such as Buchberger's criteria, improved pair selection strategies, sparse representations, or Syzygy-based optimizations. This results in a branch-heavy, allocation-intensive workload dominated by symbolic term manipulation. This produces a workload with irregular control flow and substantial abstraction pressure relative to the more numerically structured kernels.

4.2 Interfaces and Classes

The implementation of generics comes in two parts, the development of interfaces that define the capabilities of a given class and the classes that implement those interfaces. Some minor differences will appear due to the differences in capabilities between languages. For example, some languages support function overloading, whereas others do not. As such, the same function may be called different things across implementations.

4.2.1 Interfaces

Field Interface

The Field interface requires the implementing class to support the standard field operations: addition, subtraction, multiplication, and multiplicative inversion for all nonzero elements. This includes familiar examples such as the rational numbers, the complex numbers, and finite fields. As a counter-example, the set of all square matrices over a field does not itself form a field under standard matrix operations because not every nonzero matrix is invertible and multiplication is noncommutative.

For uniformity and to reduce code bloat, the interface exposes four core operation signatures: a for addition, s for subtraction, m for multiplication, and d for division, along with corresponding in-place variants (ae, se, me, de) that modify the existing object rather than returning a new value.

The implementations also provide structural utilities required by the benchmark kernels, including coercions between numeric types, constructors for the additive and multiplicative identities, and predicates for testing whether a value is zero or one. These functions support algorithms such as LU decomposition, FFTs, and Gröbner basis computations, all of which rely on consistent identity handling and type coercion.

The following is the full implementation of the Field interface for Rust that the other implementations mirror.

```rust
pub trait IField {
    fn a(&self, o: &Self) -> Self;
    fn ae(&mut self, o: &Self);
    fn s(&self, o: &Self) -> Self;
    fn se(&mut self, o: &Self);
    fn m(&self, o: &Self) -> Self;
    fn me(&mut self, o: &Self);
    fn d(&self, o: &Self) -> Self;
    fn de(&mut self, o: &Self);
    fn coerce_to_f64(&self) -> f64;
    fn coerce(&self, value: f64) -> Self;
    fn is_zero(&self) -> bool;
    fn is_one(&self) -> bool;
    fn zero(&self) -> Self;
    fn one(&self) -> Self;
}
```

Listing 1: Rust implementation of the Field interface

Math Interface

The Math interface requires the implementing class to support absolute values and square root operations. This is primarily used for partial pivoting during the LU decomposition algorithm where comparisons must be made using magnitudes rather than raw values. These operations also serve as useful utilities for validating results. For example, it is useful to compute RMS errors after conducting tests such as LU decomposition or FFTs to ensure the implementation is accurate.
The following is the full implementation of the Math interface for Rust that the other implementations mirror.

```rust
pub trait IMath{
    fn abs(&self) -> Self;
    fn sqrt(&mut self) -> Self;
}
```

Listing 2: Rust implementation of the Math interface

Ordered Interface

The Ordered interface requires the implementing class to provide a well-defined ordering through the standard comparison operations: greater than, greater than or equal, less than, less than or equal, and equality. Consistent ordering is essential for several algorithms in the suite. Partial pivoting in LU decomposition depends on comparing magnitudes or values to select stable pivots while Monte Carlo integration similarly relies on comparison operations when determining whether sampled points lie within the target region.
The following is the full implementation of the Ordered interface for Rust that the other implementations mirror.

```rust
pub trait IOrdered {
    fn lt (&self, o: &Self) -> bool;
    fn le (&self, o: &Self) -> bool;
    fn gt (&self, o: &Self) -> bool;
    fn ge (&self, o: &Self) -> bool;
    fn e (&self, o: &Self) -> bool;
}
```

Listing 3: Rust implementation of the Ordered interface

Primitive Root Interface

The Primitive Root interface requires the implementing class to support the computation of roots of unity. This is accomplished by having a *primitive root* function and a *precompute roots of unity* function. Precomputing the roots of unity avoids redundant work during the butterfly stages of the FFT.
Due to the differences in how roots of unity are calculated between the complex field and the finite field, the *primitive root* function, as well as a helper *pow* function are used in the finite-field case to *precompute roots of unity* while in the complex field case, roots of unity are obtained through trigonometric functions.

The following is the full implementation of the Primitive Root interface for Rust that the other implementations mirror.

```rust
pub trait IPrimitiveRoots<N> {
    fn primitive_root(&self, p: u64) -> N;
    fn pow(&self, exp: u64) -> N;
    fn precomputeRootsOfUnity(&self, n: u32, direction: i32) -> Vec<N>;
}
```

Listing 4: Rust implementation of the Primitive Root interface

Exponent Interface

The Exponent interface requires the implementing class to define operations on exponent vectors which represent monomials in multivariate polynomial rings. These vectors are central to Gröbner basis computations, where monomial manipulation and comparison drive the algorithm's structure. For this purpose, there are six functions in this interface: addition, subtraction, least common multiple, degree, comparison under a specified monomial ordering, and a *can_reduce* function. Both the *LCM* and *can_reduce* functions are not necessarily unique to exponent vectors but are crucial for the S-polynomials and reduction steps in the Gröbner basis algorithm. The comparison function reflects the chosen monomial order (e.g. lexicographic, graded lexicographic, or reverse lexicographic) to support different orderings as the Gröbner basis algorithm may produce different intermediate reductions and execution characteristics depending on the chosen monomial ordering.
In addition, both TypeScript and Go do not come with object equality and thus require an explicit method definition. However, Rust and Java do come with object equality and therefore do not have an explicit method in these interfaces.
The following is the full implementation of the Exponent interface for Rust that the other implementations mirror.

```rust
pub trait IExponent {
    fn add(&self, o: &Self) -> Self;
    fn sub(&self, o: &Self) -> Self;
    fn lcm(&self, other: &Self) -> Self;
    fn degree(&self) -> u32;
    fn lex_compare(&self, other: &Self) -> std::cmp::Ordering;
    fn can_reduce(&self, divisor: &Self) -> bool;
}
```

Listing 5: Rust implementation of the Exponent interface

4.2.2 Classes

The following covers details on the implementation of each class that will be used for the generic algorithms. Due to the size of each file, only the Rust versions of each class will be detailed, and sometimes with only the most relevant functions displayed. This is to keep descriptions concise and to keep focus on differences between implementations, if any. Full implementation details are provided at an external repository.
There are several differences between implementations due to differences in languages that need to be explained.
Although for all languages there is an *ICopiable* interface that simply returns a copy of the object, Rust classes also have the built-in *Clone* interface that is supported by the built-in vector struct. Additionally, Rust classes also implement the built-in *PartialEq* and *Eq* interfaces as to allow comparisons using the standard equality operator ==. Finally, hashing is implemented differently for each language. The field classes for Rust implement a Hash trait for usage with HashSets due to Rust's strict type system. This is in contrast to Java that inherits hashing from *Object*. Go has several different hashing packages that can be utilized, for the purposes of this implementation *fnv* is used for hashing. TypeScript does not provide built-in value-based hashing for complex objects in the same way as Rust or Java, requiring an alternative approach.
TypeScript only has one fixed-size numeric primitive type, namely *number*, which is a double-precision float. Unfortunately, this makes SingleField and DoubleField identical, and makes the IntModP (that is, FiniteField) class less appropriate for exact integer arithmetic, because the modulus operator (%) must operate on floating-point numbers rather than integers. For BitPackedExponent, which works explicitly with a bit string, BigInt is used.
Implications and requirements for specific design decisions will be discussed in Chapter 7.

SingleField

The SingleField class represents a single-precision floating-point number, i.e., a floating-point numeric type with 32 bits of precision. This class implements many of the aforementioned interfaces, specifically the *Field*, *Math*, and *Ordered* interfaces.
The following is the full implementation of the SingleField class for Rust that the other implementations mirror.

```
1 use std::fmt;
2 use crate::generic::i_copiable::ICopiable;
3 use crate::generic::i_field::IField;
4 use crate::generic::i_math::IMath;
```

```rust
use crate::generic::i_ordered::IOrdered;
use std::hash::Hash;
use std::cmp::Eq;
#[derive(Debug)]
pub struct SingleField {
    pub f: f32,
}

impl SingleField {
    pub fn new(f: f32) -> Self {
        SingleField { f }
    }
}

impl IField for SingleField {
    fn a(&self, o: &SingleField) -> SingleField {
        SingleField::new(self.f + o.f)
    }
    fn ae(&mut self, o: &SingleField) {
        self.f += o.f;
    }
    fn s(&self, o: &SingleField) -> SingleField {
        SingleField::new(self.f - o.f)
    }
    fn se(&mut self, o: &SingleField) {
        self.f -= o.f;
    }
    fn m(&self, o: &SingleField) -> SingleField {
        SingleField::new(self.f * o.f)
    }
    fn me(&mut self, o: &SingleField) {
        self.f *= o.f;
    }
    fn d(&self, o: &SingleField) -> SingleField {
        SingleField::new(self.f / o.f)
    }
    fn de(&mut self, o: &SingleField) {
        self.f /= o.f;
    }
    fn coerce_to_f64(&self) -> f64 {
        self.f as f64
```

```
46      }
47      fn coerce_from_int(&self, value: i32) -> Self {
48          SingleField::new(value as f32)
49      }
50      fn coerce(&self, value: f64) -> SingleField {
51          SingleField::new(value as f32)
52      }
53      fn is_zero(&self) -> bool {
54          self.f == 0.0
55      }
56      fn is_one(&self) -> bool {
57          self.f == 1.0
58      }
59      fn zero(&self) -> SingleField {
60          SingleField::new(0.0)
61      }
62      fn one(&self) -> SingleField {
63          SingleField::new(1.0)
64      }
65  }
66
67  impl ICopiable for SingleField {
68      fn copy(&self) -> Self {
69          SingleField::new(self.f)
70      }
71  }
72  impl Clone for SingleField {
73      fn clone(&self) -> Self {
74          self.copy()
75      }
76  }
77
78  impl IMath for SingleField {
79      fn abs(&self) -> SingleField {
80          SingleField::new(self.f.abs())
81      }
82      fn sqrt(&mut self) -> SingleField{
83          SingleField::new(self.f.sqrt())
84      }
85  }
86
```

```rust
impl IOrdered for SingleField {
    fn lt(&self, o: &SingleField) -> bool {
        self.f < o.f
    }
    fn le(&self, o: &SingleField) -> bool {
        self.f <= o.f
    }
    fn gt(&self, o: &SingleField) -> bool {
        self.f > o.f
    }
    fn ge(&self, o: &SingleField) -> bool {
        self.f >= o.f
    }
    fn e(&self, o: &SingleField) -> bool {
        self.f == o.f
    }
}

impl fmt::Display for SingleField {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}", self.f)
    }
}

impl Hash for SingleField {
    fn hash<H: std::hash::Hasher>(&self, state: &mut H) {
        let bits: u32 = self.f.to_bits();
        bits.hash(state);
    }
}

impl PartialEq for SingleField {
    fn eq(&self, other: &Self) -> bool {
        self.f == other.f
    }
}

impl Eq for SingleField {}
```

Listing 6: Rust implementation of the SingleField class

DoubleField

The DoubleField class follows essentially the same implementation as the preceding SingleField class, with the only difference being that the stored value is a double-precision floating-point number rather than a single-precision one-that is, 64 bits of precision instead of 32. Of course, in TypeScript there is no distinction: both classes still store the same number primitive, which always has 64 bits of precision.
The following is the full implementation of the DoubleField class for Rust that the other implementations mirror.

```
use std::fmt;
use crate::generic::i_copiable::ICopiable;
use crate::generic::i_field::IField;
use crate::generic::i_math::IMath;
use crate::generic::i_ordered::IOrdered;
use std::hash::Hash;
use std::cmp::Eq;
#[derive(Debug)]
    pub struct DoubleField {
    pub d: f64
}

impl DoubleField {
    pub fn new(d: f64) -> Self {
        DoubleField { d }
    }
}

impl IField for DoubleField {
    fn a(&self, o: &DoubleField) -> DoubleField {
        DoubleField::new(self.d + o.d)
    }
    fn ae(&mut self, o: &DoubleField) {
        self.d += o.d;
    }
    fn s(&self, o: &DoubleField) -> DoubleField {
        DoubleField::new(self.d - o.d)
    }
    fn se(&mut self, o: &DoubleField) {
        self.d -= o.d;
    }
```

```rust
    fn m(&self, o: &DoubleField) -> DoubleField {
        DoubleField::new(self.d * o.d)
    }
    fn me(&mut self, o: &DoubleField) {
        self.d *= o.d;
    }
    fn d(&self, o: &DoubleField) -> DoubleField {
        DoubleField::new(self.d / o.d)
    }
    fn de(&mut self, o: &DoubleField) {
        self.d /= o.d;
    }
    fn coerce_to_f64(&self) -> f64 {
        self.d
    }
    fn coerce_from_int(&self, value: i32) -> Self {
        DoubleField::new(value as f64)
    }
    fn coerce(&self, value: f64) -> DoubleField {
        DoubleField::new(value)
    }
    fn is_zero(&self) -> bool {
        self.d == 0.0
    }
    fn is_one(&self) -> bool {
        self.d == 1.0
    }
    fn zero(&self) -> DoubleField {
        DoubleField::new(0.0)
    }
    fn one(&self) -> DoubleField {
        DoubleField::new(1.0)
    }
}

impl ICopiable for DoubleField {
    fn copy(&self) -> Self {
        DoubleField::new(self.d)
    }
}
impl Clone for DoubleField {
```

```
73      fn clone(&self) -> Self {
74          self.copy()
75      }
76  }
77
78  impl IMath for DoubleField {
79      fn abs(&self) -> DoubleField {
80          DoubleField::new(self.d.abs())
81      }
82      fn sqrt(&mut self) -> DoubleField{
83          DoubleField::new(self.d.sqrt())
84      }
85  }
86
87  impl IOrdered for DoubleField {
88      fn lt(&self, o: &DoubleField) -> bool {
89          self.d < o.d
90      }
91      fn le(&self, o: &DoubleField) -> bool {
92          self.d <= o.d
93      }
94      fn gt(&self, o: &DoubleField) -> bool {
95          self.d > o.d
96      }
97      fn ge(&self, o: &DoubleField) -> bool {
98          self.d >= o.d
99      }
100     fn e(&self, o: &DoubleField) -> bool {
101         self.d == o.d
102     }
103 }
104
105
106 impl fmt::Display for DoubleField {
107     fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
108         write!(f, "{}", self.d)
109     }
110 }
111
112 impl Hash for DoubleField {
113     fn hash<H: std::hash::Hasher>(&self, state: &mut H) {
```

```
114         let bits: u64 = self.d.to_bits();
115         bits.hash(state);
116     }
117 }
118
119 impl PartialEq for DoubleField {
120     fn eq(&self, other: &Self) -> bool {
121         self.d == other.d
122     }
123 }
124
125 impl Eq for DoubleField {}
126
```

Listing 7: Rust implementation of the DoubleField Class

IntModP aka FiniteField

Implemented as IntModP for the purposes of FFTs, the Finite Field class models a field with a fixed, finite number of elements. The implementation assumes a prime modulus so that every nonzero element possesses a multiplicative inverse and the structure forms a field. Using a prime modulus is required in FFT contexts because it guarantees that the structure is a true field rather than a ring with zero divisors.

The finite field is of particular importance for two of the studied algorithms. First, the FFT, as finite fields is one of the two coefficient domains in this study that is capable of having roots of unity. This is important because it expands the scope of the former study as well as to investigate the performance difference between floating-point operations and integer arithmetic including the impact generics have on them. Second, the Gröbner basis, as a symbolic computing algorithm, requires exact arithmetic to execute. Floating-point arithmetic would introduce rounding errors that break the algebraic invariants the algorithm depends on. Finite fields avoid this problem entirely: all operations are exact, and values can be represented compactly using only integers for the value and the modulus. It should also be noted that the modulus is a static value and is therefore used for all instances of the class. This was to minimize storage costs for the class and reduce points of failure, though it reduces flexibility. This approach is also used for the specialized finite-field versions of each algorithm.

To detail some of the intricacies of the IntModP implementations, the following will be some of the more interesting functions in the Rust implementation of the IntModP class that the other implementations mirror.

```rust
fn mod_inverse(a: u64, p: u64) -> u64 {
    let mut a = a % p;
    let mut m = p;
    let mut x0: i64 = 0;
    let mut x1: i64 = 1;
    if m == 1 {
        return 0;
    }
    while a > 1 {
        let q = a / m;
        let t = m;
        m = a % m;
        a = t;
        let temp = x0;
        x0 = x1 - q as i64 * x0;
        x1 = temp;
    }
    if x1 < 0 {
        x1 += p as i64;
    }
    x1 as u64
}

fn d(&self, o: &IntModP) -> IntModP {
    let p = get_modulus();
    if o.i == 0 {
        panic!("Division by zero in IntModP");
    }
    else {
        let inv = mod_inverse(o.i, p);
        IntModP::new(self.i * inv)
    }
}
fn de(&mut self, o: &IntModP) {
    let p = get_modulus();
    if o.i == 0 {
        panic!("Division by zero in IntModP");
    }
    else {
        let inv = mod_inverse(o.i, p);
        self.i = (self.i * inv) % p;
```

```
    }
}

fn factorize(mut n: u64) -> Vec<u64> {
    let mut factors = Vec::new();
    let mut i = 2;
    while i * i <= n {
        if n % i == 0 {
            factors.push(i);
            while n % i == 0 {
                n /= i;
            }
        }
        i += 1;
    }
    if n > 1 {
        factors.push(n);
    }
    factors
}
fn mod_pow(base: u64, exp: u64, modulus: u64) -> u64 {
    let mut result = 1;
    let mut base = base % modulus;
    let mut exp = exp;
    while exp > 0 {
        if exp % 2 == 1 {
            result = (result * base) % modulus;
        }
        base = (base * base) % modulus;
        exp /= 2;
    }
    result
}
impl IPrimitiveRoots<IntModP> for IntModP {
    fn primitive_root(&self, n: u64) -> Self {
        let p = get_modulus();
        let factors = factorize(p as u64 - 1);
        for g in 2..p {
            let mut is_root = true;
            for &factor in &factors {
                if mod_pow(g, (p - 1) / factor as u64, p) == 1 {
```

```
                    is_root = false;
                    break;
                }
            }
            if is_root {
                return Self::new(g);
            }
        }
        Self::new(0)
    }

    fn pow(&self, exp: u64) -> IntModP {
        let p = get_modulus();
        IntModP::new(mod_pow(self.i, exp as u64, p))
    }

    fn precomputeRootsOfUnity(&self, n: u32, direction: i32) -> Vec<IntModP> {
        let p = get_modulus();
        if (p - 1) % n as u64 != 0 {
            panic!("n must divide p-1 for roots of unity to exist in IntModP");
        }
        let g = self.primitive_root(p);
        let omega = g.pow((p - 1) / (n as u64));
        let mut roots = Vec::with_capacity(n as usize);
        for k in 0..n as i32 {
            let mut exponent: u64 = ((k * direction + (p - 1) as i32) % (p - 1)
            ↪  as i32) as u64;
            if exponent < 0 {
                exponent += (p - 1) as u64;
            }
            roots.push(omega.pow(exponent));
        }
        roots
    }
}
```

Listing 8: Rust implementation of the IntModP Class

ComplexField

The ComplexField class is used to represent complex numbers with any base field. In this way, the real and imaginary parts of the complex number can be any class that implements the field interface. Although in this study the class is used only with SingleField and DoubleField as base fields, it is able to support finite fields as well.
Like the finite field implementation, this class is important because it supports roots of unity required by the FFT.
An important additional note is that the *sqrt* function is not implemented under the *IMath* interface due to it being unnecessary for the studied algorithms.
To detail some of the intricacies of the ComplexField implementations, the following excerpts show some of the more interesting functions in the Rust implementation of the ComplexField class that the other implementations mirror.

```rust
fn d(&self, o: &ComplexField<T>) -> ComplexField<T> {
    // (a + bi) / (c + di) = [(ac + bd) / (c^2 + d^2)] + [(bc - ad) / (c^2 + d^2)]i
    let denom = o.re.m(&o.re).a(&o.im.m(&o.im));
    ComplexField::new(
        self.re.m(&o.re).a(&self.im.m(&o.im)).d(&denom),
        self.im.m(&o.re).s(&self.re.m(&o.im)).d(&denom),
    )
}
fn de(&mut self, o: &ComplexField<T>) {
    let denom = o.re.m(&o.re).a(&o.im.m(&o.im));
    let temp_re = self.re.m(&o.re).a(&self.im.m(&o.im)).d(&denom);
    let temp_im = self.im.m(&o.re).s(&self.re.m(&o.im)).d(&denom);
    self.re = temp_re;
    self.im = temp_im;
}
// Implement PrimitiveRoot for SingleField numbers
impl IPrimitiveRoots<ComplexField<SingleField>> for ComplexField<SingleField> {
    fn primitive_root(&self, n: u64) -> Self {
        if n == 0 {
            panic!("n must be positive");
        }

        // Compute the angle for the primitive root of unity
        let angle = 2.0 * PI / n as f64;

        // Compute real and imaginary parts
```

```rust
        let real = SingleField::new(angle.cos() as f32);
        let imag = SingleField::new(angle.sin() as f32);

        ComplexField::new(real, imag)
    }
    fn pow(&self, exponent: u64) -> Self {
        if exponent == 0 {
            return self.one(); // Any number to the power of 0 is 1
        }
        // Convert to polar form
        let r = self.abs(); // Modulus
        let theta = self.im.coerce_to_f64().atan2(self.re.coerce_to_f64()); //
        ↪ Argument (angle)

        // Compute new modulus and argument
        let new_r = r.coerce_to_f64().powi(exponent as i32); // r^exponent
        let new_theta = theta * exponent as f64; // theta * exponent

        // Convert back to rectangular form
        let real = self.re.coerce(new_r * new_theta.cos());
        let imag = self.im.coerce(new_r * new_theta.sin());
        ComplexField::new(real, imag)
    }

    fn precomputeRootsOfUnity(&self, n: u32, direction: i32) ->
    ↪ Vec<ComplexField<SingleField>> {
        let mut roots = Vec::new();
        for k in 0..n {
            let angle = 2.0 * PI * k as f64 / n as f64 * direction as f64;
            let real = SingleField::new(angle.cos() as f32);
            let imag = SingleField::new(angle.sin() as f32);
            roots.push(ComplexField::new(real, imag));
        }
        roots
    }
}
// Implement PrimitiveRoot for DoubleField numbers
// Implement PrimitiveRoot for finite fields
```

Listing 9: Rust implementation of the ComplexField class

VecExponent

The VecExponent class represents a vector representation of the exponent vector for monomials. In this way each implementation uses its language's built-in resizable collection. This was chosen due to TypeScript's lack of fixed-size collections outside of tuples.
Each implementation is straightforward; functionally, they all simply implement the *IExponent* interface. Additionally, because the stored data type is a vector, all functions iterate over the elements of the stored vector to perform the relevant operations (e.g. add, subtract, degree, etc.)
The following is the full implementation of the VecExponent class for Rust that the other implementations mirror.

```rust
use std::fmt;
use crate::generic::i_exponent::IExponent;
use crate::generic::i_ordered::IOrdered;
use std::hash::{Hash, Hasher};

#[derive(Clone, PartialEq, Eq, Debug)]
pub struct VecExponent{
    pub exponents:  Vec<u32>
}

impl VecExponent {
    pub fn new(exponents: Vec<u32>) -> Self {
        VecExponent { exponents }
    }
}

impl Hash for VecExponent {
    fn hash<H: Hasher>(&self, state: &mut H) {
        for exp in &self.exponents {
            exp.hash(state);
        }
    }
}

impl IExponent for VecExponent {
    fn add(&self, o: &Self) -> Self {
        let result: Vec<u32> = self.exponents.iter().zip(&o.exponents).map(|(a,
        ↪  b)| a + b).collect();
        VecExponent{ exponents: result }
```

```
29      }
30
31      fn sub(&self, o: &Self) -> Self {
32          let result: Vec<u32> = self.exponents.iter().zip(&o.exponents).map(|(a,
            ↪  b)| a - b).collect();
33          VecExponent{ exponents: result }
34      }
35
36      fn lcm(&self, other: &Self) -> Self {
37          let result: Vec<u32> = self
38              .exponents
39              .iter()
40              .zip(&other.exponents)
41              .map(|(a, b)| std::cmp::max(*a, *b))
42              .collect();
43          VecExponent{ exponents: result }
44      }
45
46      fn degree(&self) -> u32 {
47          self.exponents.iter().sum()
48      }
49
50      fn lex_compare(&self, other: &Self) -> std::cmp::Ordering {
51          for (a, b) in self.exponents.iter().zip(&other.exponents) {
52              if a < b {
53                  return std::cmp::Ordering::Less;
54              } else if a > b {
55                  return std::cmp::Ordering::Greater;
56              }
57          }
58          std::cmp::Ordering::Equal
59      }
60
61      fn can_reduce(&self, divisor: &Self) -> bool {
62          for (a, b) in self.exponents.iter().zip(&divisor.exponents) {
63              if a < b {
64                  return false;
65              }
66          }
67          true
68      }
```

```rust
}


impl fmt::Display for VecExponent {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        let degree: u32 = self.degree();
        write!(f, "Degree: {:04X}, Exponents: ", degree)?;
        for exp in &self.exponents {
            write!(f, "{:02X} ", exp)?;
        }
        Ok(())
    }
}
```

Listing 10: Rust implementation of the VecExponent class

BitPackedExponent

The BitPackedExponent class utilizes a bitpacked representation of the exponent vector for monomials. Each implementation represents a vector of 6 variables using 8 bits for each exponent, with remaining last 16 bits used for the total degree of the exponents. Unfortunately, due to the lack of a 64 bit numeric type in TypeScript, an alternative was used, namely *BigInt*, an arbitrarily large numeric type for representing integers.
As with VecExponent class, the implementations are straightforward only implementing the *IExponent* interface for functionality. Importantly, because the exponent vector is now a single numeric value, this reduces many operations to a small number of integer arithmetic or bitwise operations. Addition and subtraction of the exponent vector now only requires adding or subtracting the numeric values, whereas lexicographical comparisons and retrieving the degree require bit-masking. Some operations still require iterating over all packed exponents. These include finding the least common multiple between two exponent vectors and determining if one exponent vector can reduce another (i.e. all exponents of a are greater than those of b). This was accomplished through a combination of bit-masking and bit-shifting.
The following is the full implementation of the BitPackedExponent class for Rust that the other implementations mirror.

```rust
use std::fmt;
use crate::generic::i_exponent::IExponent;
use std::hash::{Hash, Hasher};

#[derive(Clone, PartialEq, Eq, Debug)]
pub struct BitPackedExponent {
    pub exponents: u64
}
impl BitPackedExponent {
    pub fn new(exponents: u64) -> Self {
        BitPackedExponent { exponents }
    }

    pub fn from_vec(exponents: [u8; 6]) -> Self {
        let mut packed: u64 = 0;

        // Pack the exponents with the first exponent at the left (bits 47..40)
        // Layout: [63..48]=degree, [47..40]=e0, [39..32]=e1, ... [7..0]=e5
        for (i, &exp) in exponents.iter().enumerate() {
            let shift = 40_u32.saturating_sub((8 * i) as u32);
            packed |= (exp as u64) << shift;
        }

        // Compute the total degree (sum of all exponents)
        let degree: u16 = exponents.iter().map(|&e| e as u16).sum();

        // Add the degree to the top 16 bits of the u64
        packed |= (degree as u64) << 48;

        BitPackedExponent { exponents: packed }
    }
}

impl Hash for BitPackedExponent {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.exponents.hash(state);
    }
}

impl IExponent for BitPackedExponent {
    fn add(&self, o: &Self) -> Self {
```

```
        BitPackedExponent { exponents: self.exponents + o.exponents }
    }

    fn sub(&self, o: &Self) -> Self {
        BitPackedExponent { exponents: self.exponents - o.exponents }
    }

    fn lcm(&self, other: &Self) -> Self {
        let self_exponents = self.exponents & 0x0000_FFFF_FFFF_FFFF;
        let other_exponents = other.exponents & 0x0000_FFFF_FFFF_FFFF;

        let mut lcm_exponents: u64 = 0;
        let mut degree: u16 = 0;

        for i in (0..48).step_by(8) {
            let self_exp = (self_exponents >> i) & 0xFF;
            let other_exp = (other_exponents >> i) & 0xFF;

            let lcm_exp = self_exp.max(other_exp);

            lcm_exponents |= lcm_exp << i;
            degree += lcm_exp as u16; // Accumulate the degree
        }

        // Set the degree in the top 16 bits
        lcm_exponents |= (degree as u64) << 48;

        BitPackedExponent { exponents: lcm_exponents }
    }

    fn degree(&self) -> u32 {
        ((self.exponents >> 48) & 0xFFFF) as u32
    }

    fn lex_compare(&self, other: &Self) -> std::cmp::Ordering {
        for i in (0..48).step_by(8).rev() {
            let self_exp = self.exponents & 0x0000_FFFF_FFFF_FFFF;
            let other_exp = other.exponents & 0x0000_FFFF_FFFF_FFFF;

            if self_exp < other_exp {
                return std::cmp::Ordering::Less;
```

```rust
            } else if self_exp > other_exp {
                return std::cmp::Ordering::Greater;
            }
        }
        std::cmp::Ordering::Equal
    }

    fn can_reduce(&self, divisor: &Self) -> bool {
        for i in (0..48).step_by(8) {
            let self_exp = (self.exponents >> i) & 0xFF;
            let divisor_exp = (divisor.exponents >> i) & 0xFF;

            if self_exp < divisor_exp {
                return false;
            }
        }
        true
    }

}


impl fmt::Display for BitPackedExponent {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        // Extract degree (top 16 bits)
        let degree = (self.exponents >> 48) & 0xFFFF;
        write!(f, "Degree: {:04X}, Exponents (hex): ", degree)?;
        // Print each 8-bit exponent in hex, left to right (e0..e5)
        for i in (0..48).step_by(8).rev() {
            let exp = (self.exponents >> i) & 0xFF;
            write!(f, "{:02X} ", exp)?;
        }
        Ok(())
    }
}
```

Listing 11: Rust implementation of the BitPackedExponent class

4.3 Implementations

4.3.1 Restrictions

Implementations of each algorithm are the result of many design iterations due to the many differences in language capabilities as well as the desire for a fair, naive, and language-agnostic design. This leads to some slight differences among implementations that will be discussed in further detail in Chapter 7.

Linear Congruential Generator

For the sake of reproducibility and consistent inputs, rather than using each language's built-in random number generators, a Linear Congruential Generator (LCG) was implemented in each language and used to generate values for the relevant test cases. The generator produces pseudo-random numbers through the recurrence

$$seed = (a \cdot seed + c) \mod m$$

where a, c, and m are constant parameters. The parameters were selected primarily to provide deterministic and reproducible sequences using arithmetic behaviour that could be implemented consistently across all evaluated languages and runtime environments. In particular, the implementation used $a = 16645$, $m = 1345$, $c = 1013904$, and an initial seed of 12345. The LCG was not intended to provide high-quality statistical randomness or rigorous Monte Carlo sampling properties, but rather a portable and reproducible source of pseudo-random inputs for benchmarking purposes. Consequently, the Monte Carlo benchmark should be interpreted primarily as a computational workload representative rather than as a statistically rigorous stochastic simulation.
The following is the Rust implementation of the LCG class that the other implementations mirror.

```rust
pub struct Lcg {
    m: i32,
    a: i32,
    c: i32,
    last_num: i32,
}

impl Lcg {
    pub fn new(seed: i32, m: i32, a: i32, c: i32) -> Self {
        Self {
```

```
            m,
            a,
            c,
            last_num: seed,
        }
    }

    pub fn next_int(&mut self) -> i32 {
        self.last_num = (self.a * self.last_num + self.c) % self.m;
        self.last_num
    }

    pub fn next_double(&mut self) -> f64 {
        self.next_int() as f64 / self.m as f64
    }
}
```

Listing 12: Rust implementation of the LCG class

4.3.2 Monte Carlo Integration

Monte Carlo integration is straightforward to implement. Following the description as detailed in chapter 3, and using the implemented LCG, the algorithm generates a pair of random floating–point numbers, x and y between 0 and 1, inclusive. Taking this pair of points as a point in space, the squared distance from the origin, (0,0), is calculated by taking $distance = x^2 + y^2$. If this distance is less than or equal to 1 then it lies within the bounds of a quarter unit circle and a counter is incremented. Finally, by taking the ratio of the number of points that landed within the bounds of the quarter unit circle to the total number of samples taken and multiplying by four, this gives an estimate of π that improves with more samples and with a sufficiently high period for the LCG.
The following is the full implementation of the specialized f64 Monte Carlo class in Rust that the other implementations mirror.

```
use crate::helpers::lcg::Lcg;

fn integrate(num_samples: usize) -> f64 {
    let mut rng = Lcg::new(12345, 1345, 16645, 1013904);
    let mut under_curve = 0;
    for _ in 0..num_samples {
```

```
        let x: f64 = rng.next_double();
        let y: f64 = rng.next_double();
        if x * x + y * y <= 1.0 {
            under_curve += 1;
        }
    }
    (under_curve as f64 / num_samples as f64) * 4.0
}

fn main() {
    let args: Vec<String> = std::env::args().collect();
    let mut num_samples = 1_000_000;
    if args.len() > 1 {
        num_samples = args[1].parse().unwrap_or(num_samples);
    }

    let pi = integrate(num_samples);
    println!("Pi is approximately: {}", pi);
    println!("Num samples: {}", num_samples);
    println!("RMS Error: {}", (std::f64::consts::PI - pi).abs());
}
```

Listing 13: Rust implementation of Monte Carlo

4.3.3 Successive Over-Relaxation

The Successive Over-Relaxation (SOR) algorithm is also quite simple to implement. Each iteration of the algorithm applies a weighted 5-point averaging stencil over the grid with a relaxation factor ω to accelerate convergence. In these implementations the grid starts at a state of all 0s except for the top edge that starts at 100. Additionally, the grid is always an *n-by-n* square, ω is always 1.5, or equivalent, and the SOR algorithm is run 10,000 times. For the generic implementation, because the algorithm only requires that the provided type implement the *IField* interface, finite fields would be supported although no practical SOR application would use them because finite fields do not provide a meaningful notion of geometric proximity or convergence.
The following is the full implementation of the specialized f64 Successive Over-Relaxation class in Rust that the other implementations mirror.

```rust
pub fn execute(omega: f64, g: &mut Vec<Vec<f64>>, num_iterations: usize) {
    let m = g.len();
    let n = g[0].len();

    let omega_over_four = omega * 0.25;
    let one_minus_omega = 1.0 - omega;

    let mm1 = m - 1;
    let nm1 = n - 1;

    for _ in 0..num_iterations {
        for i in 1..mm1 {
            let gim1 = g[i - 1].clone();
            let gip1 = g[i + 1].clone();
            for j in 1..nm1 {
                g[i][j] = omega_over_four * (gim1[j] + gip1[j] + g[i][j - 1] +
                ↪  g[i][j + 1])
                    + one_minus_omega * g[i][j];
            }
        }
    }
}

pub fn print_matrix(a: &Vec<Vec<f64>>) {
    for row in a {
        for val in row {
            print!("{:.2} ", val);
        }
        println!();
    }
}

fn main() {
    // arg1 = grid size n (nxn)
    let args = std::env::args().collect::<Vec<String>>();
    let n: usize = args.get(1).and_then(|s| s.parse().ok()).unwrap_or(16);
    let m = n;
    let num_iterations = 10000;
    let omega = 1.5;
    let mut g = vec![vec![0.0; n]; m];

```

```rust
    // Set boundary conditions

    for j in 0..n {
        g[0][j] = 100.0;        // Top edge (hot)
    }

    println!("Rust specialized double SOR");
    println!("Grid size: {}x{}", m, n);
    println!("Number of iterations: {}", num_iterations);
    execute(omega, &mut g, num_iterations);

}
```

Listing 14: Rust implementation of Successive Over-Relaxation

4.3.4 LU Factorization

The LU factorization algorithm computes a dense in-place decomposition of a given square matrix using classical Gaussian elimination with partial pivoting. This implementation both factors the matrix as described previously and solves for the vector x satisfying $Ax = b$. Matrix A is generated to ensure diagonal dominance by keeping a running total of each row's values and summing it to the diagonal value. The vector b is generated randomly. Both are generated using a different LCG with parameters $a = 16645, m = 2^{13} - 1, c = 1013904$ with an initial value, or seed, of 12345. A relatively large prime modulus was chosen to increase the likelihood that randomly generated matrices over the finite field would be nonsingular.
The following is the full implementation of the specialized f64 LU factorization class in Rust that the other implementations mirror.

```rust
use crate::helpers::lcg::Lcg;

fn print_matrix(a: &Vec<Vec<f64>>) {
    for row in a {
        for val in row {
            print!("{:.3} ", val);
        }
        println!();
    }
}
```

```rust

fn print_vector(b: &Vec<f64>) {
    for val in b {
        print!("{:.3} ", val);
    }
    println!();
}

pub fn factor(a: &mut Vec<Vec<f64>>, pivot: &mut Vec<usize>) -> i32 {
    let n = a.len();
    let m = a[0].len();
    let min_mn = std::cmp::min(m, n);

    for j in 0..min_mn {
        // Find pivot in column j and test for singularity
        let mut jp = j;
        let mut t = a[j][j].abs();
        for i in (j + 1)..m {
            let ab = a[i][j].abs();
            if ab > t {
                jp = i;
                t = ab;
            }
        }
        pivot[j] = jp;

        // If zero pivot, factorization fails
        if a[jp][j] == 0.0 {
            return 1;
        }

        // Swap rows j and jp if needed
        if jp != j {
            a.swap(j, jp);
        }

        // Compute elements j+1:M of jth column
        if j < m - 1 {
            let recp = 1.0 / a[j][j];
            for k in (j + 1)..m {
                a[k][j] *= recp;
```

```
            }
        }

        // Rank-1 update to trailing submatrix
        if j < min_mn - 1 {
            for ii in (j + 1)..m {
                let aii_j = a[ii][j];
                for jj in (j + 1)..n {
                    a[ii][jj] -= aii_j * a[j][jj];
                }
            }
        }
    }
    0
}

pub fn solve(lu: &Vec<Vec<f64>>, pvt: &Vec<usize>, b: &mut Vec<f64>) {
    let m = lu.len();
    let n = lu[0].len();
    let mut ii = 0usize;
    for i in 0..m {
        let ip = pvt[i];
        let mut sum = b[ip];
        b[ip] = b[i];
        if ii == 0 {
            for j in ii..i {
                sum -= lu[i][j] * b[j];
            }
        } else if sum == 0.0 {
            ii = i;
        }
        b[i] = sum;
    }

    for i in (0..n).rev() {
        let mut sum = b[i];
        for j in (i + 1)..n {
            sum -= lu[i][j] * b[j];
        }
        b[i] = sum / lu[i][i];
    }
```

```rust
}

fn main() {
    let args: Vec<String> = std::env::args().collect();
    let mut n = 4;
    if args.len() > 1 {
        n = args[1].parse().unwrap_or(4);
    }
    let mut rand = Lcg::new(12345, 1345, 16645, 1013904);
    let mut a: Vec<Vec<f64>> = vec![vec![0.0; n]; n];
    for i in 0..n {
        let mut row_sum = 0.0;
        for j in 0..n {
            if i != j {
                let val = rand.next_double() * 1000.0;
                a[i][j] = val;
                row_sum += val.abs();
            }
        }
        // Set diagonal to be strictly greater than row_sum
        a[i][i] = row_sum + rand.next_double() * 1000.0 + 1.0;
    }
    let mut b: Vec<f64> = (0..n).map(|_| rand.next_double() * 1000.0).collect();
    println!("Rust specialized double LU");
    println!("Matrix size: {}", n);
    for i in 0..10 {
        let mut pivot: Vec<usize> = vec![0; n];
        let mut a_copy = a.clone();
        let mut b_copy = b.clone();
        let result = factor(&mut a_copy, &mut pivot);
        if (result == 1) {
            println!("Factorization failed due to singularity");
            continue;
        }
        solve(&a_copy, &pivot, &mut b_copy);
        println!("Iteration {} completed", i);
    }
}
```

Listing 15: Rust implementation of LU factorization

4.3.5 Fast Fourier Transform

The Fast Fourier Transform benchmark performs a one-dimensional forward transform of a fixed-size vector using the iterative radix-2 Cooley-Tukey algorithm. In this study, there are two specialized implementations of the algorithm for complex coefficients and finite-field coefficients. Once again, the input values are randomly generated using the implemented LCG. The size of the vector is passed in as an argument and is determined by the test case used. Each iteration of the algorithm performs both a forward transform and an inverse transform on the input data, providing an opportunity to validate results and the algorithm if necessary. In addition, as a way to validate correctness on the finite-field version, specific input values were compared against known outputs to test the implementation.
As previously mentioned, the FFT requires the coefficient domain to have a principal root of unity and this means the two implementations will differ at specific steps of the transformation algorithm.
The following will focus only on the major differences between the two different specialized implementations of the FFT in Rust.

```
fn transform_internal(data: &mut [f64], direction: i32) {
        if data.is_empty() {
            return;
        }

        let n = data.len() / 2; // Number of complex elements
        if n == 1 {
            return; // Identity operation
        }

        let logn = Self::log2(n);

        // Bit-reverse the input data for decimation-in-time algorithm
        Self::bitreverse(data);

        // Apply FFT recursion
        for bit in 0..logn {
            let dual = 1 << bit; // 2^bit
            let mut w_real = 1.0;
            let mut w_imag = 0.0;

            let theta = 2.0 * direction as f64 * std::f64::consts::PI / (2.0 *
            ↪  dual as f64);
```

```
        let s = theta.sin();
        let t = (theta / 2.0).sin();
        let s2 = 2.0 * t * t;

        // a = 0
        for b in (0..n).step_by(2 * dual) {
            let i = 2 * b;
            let j = 2 * (b + dual);

            let wd_real = data[j];
            let wd_imag = data[j + 1];

            data[j] = data[i] - wd_real;
            data[j + 1] = data[i + 1] - wd_imag;
            data[i] += wd_real;
            data[i + 1] += wd_imag;
        }

        // a = 1 .. (dual-1)
        for a in 1..dual {
            // Trigonometric recurrence for w -> exp(i * theta) * w
            let tmp_real = w_real - s * w_imag - s2 * w_real;
            let tmp_imag = w_imag + s * w_real - s2 * w_imag;
            w_real = tmp_real;
            w_imag = tmp_imag;

            for b in (0..n).step_by(2 * dual) {
                let i = 2 * (b + a);
                let j = 2 * (b + a + dual);

                let z1_real = data[j];
                let z1_imag = data[j + 1];

                let wd_real = w_real * z1_real - w_imag * z1_imag;
                let wd_imag = w_real * z1_imag + w_imag * z1_real;

                data[j] = data[i] - wd_real;
                data[j + 1] = data[i + 1] - wd_imag;
                data[i] += wd_real;
                data[i + 1] += wd_imag;
            }
```

```
            }
            //println!("{} {}", data[bit], data[bit + 1]);
        }
    }
```

Listing 16: Excerpts of the Rust implementation of FFT with complex fields

```
fn mod_inverse(a: i32, m: i32) -> i32 {
    let mut m = m;
    let mut a = a;
    let (mut x0, mut x1) = (0, 1);
    let m0 = m;
    while a > 1 {
        let q = a / m;
        let t = m;
        m = a % m;
        a = t;
        let t = x0;
        x0 = x1 - q * x0;
        x1 = t;
    }
    if x1 < 0 { x1 += m0; }
    x1
}

fn modpow(mut base: i32, mut exp: i32) -> i32 {
    let modulus = unsafe {MODULUS};
    if modulus == 0 { panic!("Modulus must be positive"); }
    let mut result = 1;
    base %= modulus;
    while exp > 0 {
        if exp % 2 == 1 {
            result = (result * base) % modulus;
        }
        base = (base * base) % modulus;
        exp /= 2;
    }
    result
}
fn primitive_root() -> i32 {
```

```
    let modulus = unsafe {MODULUS};
    fn factorize(mut n: i32) -> Vec<i32> {
        let mut factors = Vec::new();
        let mut i = 2;
        while i * i <= n {
            if n % i == 0 {
                factors.push(i);
                while n % i == 0 {
                    n /= i;
                }
            }
            i += 1;
        }
        if n > 1 {
            factors.push(n);
        }
        factors
    }
    let p = modulus;
    let factors = factorize (p - 1);
    for g in 2..p {
        let mut is_root = true;
        for &factor in &factors {
            if modpow(g, (p - 1) / factor) == 1 {
                is_root = false;
                break;
            }
        }
        if is_root {
            return g;
        }
    }
    0

}
fn precomputeRootsOfUnity(n: i32, direction: i32) -> Vec<i32> {
    let modulus = unsafe {MODULUS};
    if (modulus - 1) % n != 0 {
        panic!("n must divide p-1 for roots of unity to exist in IntModP");
    }
    let g = primitive_root();
```

```
75      let omega = modpow(g, (modulus-1)/n);
76      let mut roots = Vec::with_capacity(n as usize);
77      for k in 0..n {
78          let mut exponent = (k * direction % (modulus - 1));
79          if exponent < 0 {
80              exponent += (modulus - 1);
81          }
82          roots.push(modpow(omega, exponent));
83      }
84      roots
85  }
86  fn transform_internal(data: &mut [i64], direction: i32) {
87      let modulus = unsafe {MODULUS};
88      if data.is_empty() {
89          return;
90      }
91
92      let n = data.len(); // Now n is the length of the data array (no division by
        ↪ 2)
93      if n == 1 {
94          return; // Identity operation
95      }
96
97      let logn = Self::log2(n);
98      Self::bitreverse(data);
99
100     let roots = precomputeRootsOfUnity(n as i32, direction);
101
102     let mut dual = 1;
103     for _bit in 0..logn {
104         for a in 0..dual {
105             let w = roots[a * (n / (2 * dual))].clone(); // Use precomputed root
106             for b in (0..n).step_by(2 * dual) {
107                 let i = b + a;
108                 let j = b + a + dual;
109
110                 let wd = w as i64 * &data[j] % modulus as i64; // Twiddle factor
                    ↪ multiplication
111                 data[j] = (data[i] + modulus as i64 - wd) % modulus as i64; //
                    ↪ Subtract
112                 data[i] = (data[i] + wd) % modulus as i64; // Add
```

```
113                 }
114             }
115             dual *= 2;
116         }
117     }
```

Listing 17: Excerpts of the Rust implementation of FFT with finite fields

4.3.6 Gröbner Basis

The Gröbner basis kernel implements a naive form of Buchberger's algorithm for multivariate polynomial ideals. Specifically, the implementation intentionally omits major optimizations commonly used in practical Gröbner basis systems. In detail, the algorithm begins by creating a list of index pairs that still need to be assessed. Each pair of indices is then removed from the list and the S-polynomial with the corresponding polynomials is calculated and then reduced with respect to the current basis. If this remainder is non-zero and does not already exist in the basis it is added to the basis. With a new polynomial added to the basis, the size of the basis increases and new pairs of polynomials corresponding to this new polynomial are added to the list of pairs. Once there are no more pairs to assess, the final basis is reduced by itself to remove irrelevant polynomials and the final reduced basis is returned.

In this study, there are two specialized implementations of the Gröbner basis using two different representations of the vector exponent for the terms, one is a vector of numeric types and the other being a bitpacked representation using a single numeric type. Although this can lead to a notable difference in performance whenever exponents need to be compared, the difference in implementation does not change substantially as it simply replaces looping over elements in a vector with a single operation. As such, the exact details on implementation differences will be omitted here and can be seen at the external repository.

For both implementations, because of the complexity of the algorithm, validation of the results was important, especially because these implementations use a finite-field coefficient. As such, two different reference implementations were utilized to validate the results, namely Python's SymPy library and Maple. These make it possible to validate the results of the cyclic test cases, even for a finite field with any prime modulus.

The following is the Rust implementation of the Gröbner basis over the finite field with some details omitted for brevity.

```rust
pub struct Term {
    pub coefficient: u64,
    pub exponents: Vec<usize>, // Exponents for each variable
}
pub enum TermOrder {
    Lex,
    GrLex,
    RevLex
}
impl Term {
    pub fn compare(&self, other: &Term) -> std::cmp::Ordering {
        let order = unsafe { TERM_ORDER };
        match order {
            TermOrder::Lex => self.exponents.cmp(&other.exponents), // Lexicographic order
            TermOrder::GrLex => { // Graded lexicographic order
                let self_degree: usize = self.exponents.iter().sum();
                let other_degree: usize = other.exponents.iter().sum();
                self_degree.cmp(&other_degree).then_with(|| self.exponents.cmp(&other.exponents))
            }
            TermOrder::RevLex => { // Reverse lexicographic order
                let self_degree: usize = self.exponents.iter().sum();
                let other_degree: usize = other.exponents.iter().sum();
                self_degree
                    .cmp(&other_degree)
                    .then_with(|| self.exponents.iter().rev().cmp(other.exponents.iter().rev()))
            }
        }
    }
}
impl Hash for Term {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.coefficient.hash(state);
        self.exponents.hash(state);
    }
}
impl Hash for Polynomial {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.terms.hash(state);
```

```
    }
}
impl Polynomial {
    pub fn new(mut terms: Vec<Term>) -> Self {
        terms.sort_by(|a, b| b.compare(a));
        terms.retain(|t| t.coefficient != 0); // Remove zero coefficient terms
        Polynomial { terms }
    }
    pub fn reduce(&self, divisors: &[Polynomial]) -> Polynomial {
        let modulus = unsafe { MODULUS };
        let mut result = self.clone(); // Start with the input polynomial
        let mut remainder: Vec<Term> = Vec::new();

        loop {
            let mut reduced = false;

            // Iterate over the divisors to reduce the leading term
            for divisor in divisors {
                if let Some(leading_term) = result.terms.first() {
                    if let Some(divisor_leading_term) = divisor.terms.first() {
                        // Check if the leading term can be reduced
                        if
                        ↪  leading_term.exponents.iter().zip(&divisor_leading_term.exponents
                        ↪  b)| a >= b) {


                            // Compute the reduction factor
                            let coefficient = (leading_term.coefficient as u64 *
                            ↪  mod_inverse(divisor_leading_term.coefficient,
                            ↪  modulus) % modulus) as u64;
                            let exponents: Vec<usize> = leading_term
                                .exponents
                                .iter()
                                .zip(&divisor_leading_term.exponents)
                                .map(|(a, b)| a - b)
                                .collect();

                            // Create the reduction term
                            let reduction_term = Term {
                                coefficient,
                                exponents,
```

```
76                              };
77
78
79                              // Subtract the scaled divisor from the result
80                              let scaled_divisor =
                                ↪  divisor.multiply_by_term(&reduction_term);
81                              result = result.subtract(&scaled_divisor);
82
83                              reduced = true;
84                              break; // Restart the loop after reducing
85                          }
86                      }
87                  }
88              }
89
90              // If no reduction was performed, break the loop
91              if !reduced {
92                  if let Some(leading_term) = result.terms.first().cloned() {
93                      remainder.push(leading_term);
94                      result.terms.remove(0);
95                  }
96                  else {
97                      break;
98                  }
99              }
100         }
101
102
103         result.terms.append(&mut remainder);
104         Polynomial::new(result.terms)
105     }
106     pub fn s_polynomial(p1: &Polynomial, p2: &Polynomial) -> Polynomial {
107         let modulus = unsafe { MODULUS };
108         // Compute the LCM of the leading monomials' exponents
109         let mut lcm_exponents = vec![0; p1.terms[0].exponents.len()];
110         for i in 0..lcm_exponents.len() {
111             lcm_exponents[i] =
                ↪  p1.terms[0].exponents[i].max(p2.terms[0].exponents[i]);
112         }
113
114         // Compute exponent shifts for each polynomial
```

```
115         let shift_p1 =
            ↪  lcm_exponents.iter().zip(&p1.terms[0].exponents).map(|(lcm, exp)|
            ↪  lcm - exp).collect::<Vec<_>>();
116         let shift_p2 =
            ↪  lcm_exponents.iter().zip(&p2.terms[0].exponents).map(|(lcm, exp)|
            ↪  lcm - exp).collect::<Vec<_>>();
117
118         // Compute scaling factors for coefficients (modular inverses)
119         let a = p1.terms[0].coefficient;
120         let b = p2.terms[0].coefficient;
121
122
123         // S = (b * x^shift_p1 * p1 - a * x^shift_p2 * p2) mod modulus
124         let scaled_p1 = Polynomial::new(
125             p1.terms.iter().map(|term| Term {
126                 coefficient: (b * term.coefficient) % modulus,
127                 exponents: term.exponents.iter().zip(&shift_p1).map(|(exp,
                    ↪  shift)| exp + shift).collect(),
128             }).collect()
129         );
130         let scaled_p2 = Polynomial::new(
131             p2.terms.iter().map(|term| Term {
132                 coefficient: (a * term.coefficient) % modulus,
133                 exponents: term.exponents.iter().zip(&shift_p2).map(|(exp,
                    ↪  shift)| exp + shift).collect(),
134             }).collect()
135         );
136         scaled_p1.subtract(&scaled_p2)
137     }
138
139 }
140 pub fn naive_grobner_basis(polynomials: Vec<Polynomial>) -> Vec<Polynomial> {
141     let mut basis = polynomials.clone();
142     let mut basis_set: HashSet<Polynomial> = HashSet::new();
143     for poly in &basis {
144         basis_set.insert(poly.clone());
145     }
146
147     let mut pairs = Vec::<(usize, usize)>::new();
148     for i in 0..basis.len() {
149         for j in i + 1..basis.len() {
```

```
            pairs.push((i, j));
        }
    }
    while pairs.is_empty() == false {
        let (i, j) = pairs.remove(0);
        let s_poly = Polynomial::s_polynomial(&basis[i], &basis[j]);
        let reduced = s_poly.reduce(&basis);
        if !reduced.terms.is_empty() && !basis_set.contains(&reduced) {
            basis_set.insert(reduced.clone());
            let new_idx = basis.len();
            basis.push(reduced);
            pairs.extend((0..new_idx).map(|k| (k, new_idx)));
        }
    }

    //reduce basis by self
    let mut reduced_basis = Vec::new();
    for poly in &basis {
        // reduce poly by basis excluding itself
        let mut basis_excluding_self = basis.clone();
        basis_excluding_self.retain(|p| p != poly);
        let reduced = poly.reduce(&basis_excluding_self);
        if !reduced.terms.is_empty() && !reduced_basis.contains(&reduced) {
            reduced_basis.push(reduced.make_monic());
        }
    }
    reduced_basis
}

```

Listing 18: Rust implementation of Gröbner basis over the finite field

Chapter 5

Platforms

For the sake of reproducibility and to provide context for discussing both performance and language features, this chapter describes the language versions and compiler versions used for each platform.

5.1 Language and Compiler Versions

Rust

For this study, the versions of Rust and Cargo used were rustc 1.92.0 and cargo 1.92.0, both released on December 11, 2025[28].
Parametric polymorphism has been a core part of Rust since its inception. The archived Rust 1.0.0 documentation includes a dedicated section on generic data types and traits, demonstrating that generics were part of the language from the outset when version 1.0.0 was released in May 2015[29]. Since then, the language has undergone more than ninety stable releases, many of which expanded the expressiveness of the type system and generic abstractions.
Several later versions introduced important improvements to Rust's type system and generics capabilities. Rust 1.26, released in April 2018, introduced the *impl Trait* syntax that allows functions to return opaque types while retaining static dispatch and compile-time type information[30]. Rust 1.51.0, released on March 25, 2021, introduced the minimal viable product of const generics, enabling generic parameters based on constant values[31]. Rust 1.65.0, released on October 28, 2022, introduced generic associated types (GATs), allowing associated types within traits to themselves be generic[32]. These additions expanded Rust's ability to express complex generic abstractions while maintaining strong compile-time guarantees.

Java

Java benchmarks were executed using Microsoft OpenJDK 25.0.1 LTS (Server VM, mixed mode), including the javac 25.0.1 compiler, released on October 21, 2025[33], based on the Java SE 25 platform originally released on September 16, 2025[34].
Generics were introduced to Java in Java 5, released on September 30, 2004 [35]. Java implements generics through type erasure, preserving backward compatibility at the cost of runtime type specialization. This release added parametric polymorphism to the language along with several other major features including enhanced for-loops, autoboxing and unboxing, and type-safe enumerations. The introduction of generics significantly improved type safety in Java collections and APIs by allowing compile-time type checking while maintaining compatibility with existing code.
Subsequent versions of Java introduced improvements to the usability of generics. Java 7 introduced the diamond operator (<>), which allows the compiler to infer generic type arguments when constructing objects[36]. Java 8 expanded type inference capabilities and introduced lambda expressions and the streams API, both of which interact extensively with generic types[37]. Later versions of Java continued to refine the language and runtime, though the core generics model introduced in Java 5 has remained largely unchanged.

Go

All Go benchmarks were compiled and executed using Go 1.25.5 (linux/amd64), a security point release from the Go 1.25 branch originally released on August 12, 2025[38] and 1.25.5 was released on December 2, 2025[39].
Generics were heavily anticipated and were added to Go with version 1.18 on March 15, 2022[40], 13 years after the language's creation in 2009[41]. Because Go adopted a deliberately minimal generics design, relatively few subsequent releases introduced major changes to the generics system. Go 1.20 refined the constraint system, including improvements to the behaviour of the comparable constraint, improving the usability of generics[42]. Go 1.21 expanded the use of generics within the standard library, including the addition of generic helper packages such as slices, maps, and cmp[43]. Thus, while there have been updates to the language and expansions on the usability of generics, the core design introduced in Go 1.18 remains largely unchanged in later versions.

TypeScript

JavaScript and TypeScript benchmarks were implemented in TypeScript 5.9.3 and executed using Node.js v24.12.0 on linux/amd64. TypeScript provides support for generic programming through parametric polymorphism. However, these types exist only at com-

pile time and are erased during compilation to JavaScript. As a result, TypeScript generics do not directly influence the runtime behaviour of the program. All benchmark execution is therefore performed by the V8 JavaScript engine included with Node.js, and performance characteristics reflect the behaviour of the JavaScript runtime rather than the TypeScript type system.
TypeScript 5.9 was released on August 1, 2025[44] with 5.9.3 being published to npm in October 2025[45], whereas Node 24.12.0 was first released on May 6, 2025, but last updated on December 10, 2025[46]. Generics have been part of TypeScript since TypeScript 1.0, released in April of 2014[47]. There have been several important updates to the type system in TypeScript that improve the expressiveness of generics, including TypeScript 2.1 which introduced mapped types allowing the transformation of one type to another by iterating over properties[48], TypeScript 2.8, which introduces conditional types[49], and TypeScript 4.1, which introduced template literal types[50]. Nevertheless, none of these are relevant for this study and have no impact on performance since types are erased at runtime.

C++

C++ benchmarks were compiled using g++ (GCC) version 11.4.0 with the `-O3` optimization flag and the `-std=c++17` language standard. The GNU Compiler Collection provides both the compiler and the associated standard library implementation used in this study. Generic programming in C++ is primarily implemented through templates, which have been part of the language since the C++98 standard released in 1998[27]. Templates enable compile-time code generation, allowing type-specific optimizations without introducing runtime overhead.
Subsequent revisions of the language improved the expressiveness of templates. C++11 introduced variadic templates, enabling more flexible generic abstractions[51]. Later standards such as C++14 and C++17 refined template metaprogramming and introduced features like `if constexpr`, which allows compile-time branching based on type information[52][53]. C++17 was selected because it provides mature template metaprogramming support while remaining widely supported across contemporary compiler toolchains. While newer standards add further features, C++17 provides a stable and representative baseline for this study.

Julia

Julia benchmarks were executed using Julia version 1.12.4, released on January 7, 2026, using the included LLVM compiler [54].

Parametric polymorphism is a core feature of Julia and is integrated with its multiple dispatch system. Since its initial release, the language has supported generic functions that are specialized based on the types of their arguments.
Julia differs from statically compiled languages in that it uses just-in-time (JIT) compilation to generate optimized machine code for concrete type combinations at runtime. This allows the compiler to specialize functions based on actual usage while maintaining a high-level programming model.
Subsequent versions of Julia have focused on improving compiler performance, type inference, and method specialization. However, the core model of parametric types and multiple dispatch has remained largely unchanged since Julia 1.0, released on August 8, 2018, and is representative for this study [55].

5.2 System Configuration

All benchmarks were run on a Framework laptop running Nobara Linux version 43 (KDE Plasma Desktop Edition), specifically kernel version 6.18.9-201.nobara.fc43.x86_64. The hardware consisted of a 13th Gen Intel(R) Core(TM) i7-1370P CPU with 14 cores (6 performance cores and 8 efficiency cores) and 20 hardware threads, along with 62 GiB of RAM.

Chapter 6

Results

This chapter presents the performance results of the benchmark suite described in Chapter 4. For each algorithm, both specialized and generic implementations were evaluated across all platforms.
To improve consistency and reduce measurement variability, all benchmarks were executed under controlled conditions, with fixed hardware settings and no background processes. For algorithms that naturally terminate after a single execution, multiple iterations were performed over the same input data to reduce measurement noise and account for runtime effects such as just-in-time (JIT) compilation. In particular, LU factorization, Fast Fourier Transform, and Naive Gröbner basis benchmarks were executed ten times per run. For Monte Carlo Integration and Successive Over-Relaxation, a large number of iterations were performed within a single execution to similarly amortize runtime overhead.
All programs were built or executed using the default high-optimization configuration appropriate to each platform, and execution time was measured using the `time` utility in a Bash environment to maintain consistency across platforms.
Platform-specific compilation and execution procedures are summarized below:

- **Rust:** Compiled using `cargo build --release`, enabling full compiler optimizations via `rustc`. Executables were run directly from the `target/release` directory.
- **Java:** Compiled using `javac` across all source files. Programs were executed using the OpenJDK Server JVM with an increased heap size (`-Xmx27G`) to avoid memory exhaustion during large benchmarks.
- **Go:** Compiled using `go build`, producing a standalone binary. Due to Go's executable structure, individual benchmarks were compiled separately by modifying the active `main` entry point prior to compilation.

- **TypeScript:** Transpiled to JavaScript using `tsc`, then executed using Node.js via the `node` runtime.
- **C++:** Compiled using `g++ -O3 -std=c++17`, enabling aggressive compiler optimizations. Executables were run directly.
- **Julia:** Executed directly using the `julia` runtime, with just-in-time compilation occurring during execution.

All benchmarks were executed on an otherwise idle system to minimize interference from background processes.

6.1 Monte Carlo Integration

Table 6.1: Monte Carlo Integration Performance for Rust

Language	Test	Field	Variant	Real Time (s)	User Time (s)	Real Speedup
Rust	1,000,000	Double	Specialized	0.026	0.020	
			Generic	0.028	0.022	0.93
	1,000,000,000	Double	Specialized	11.782	11.767	
			Generic	11.889	11.883	0.99
	2,147,483,647	Double	Specialized	25.305	25.298	
			Generic	25.278	25.271	1.00

Table 6.2: Monte Carlo Integration Performance for Java

Language	Test	Field	Variant	Real Time (s)	User Time (s)	Real Speedup
Java	1,000,000	Double	Specialized	0.078	0.059	
			Generic	0.092	0.094	0.82
	1,000,000,000	Double	Specialized	11.760	11.754	
			Generic	11.784	11.774	0.99
	2,147,483,647	Double	Specialized	25.196	25.195	
			Generic	25.201	25.199	1.00

Table 6.3: Monte Carlo Integration Performance for Go

Language	Test	Field	Variant	Real Time (s)	User Time (s)	Real Speedup
Go	1,000,000	Double	Specialized	0.043	0.034	
			Generic	0.032	0.022	1.34
	1,000,000,000	Double	Specialized	12.415	12.428	
			Generic	12.615	12.632	0.98
	2,147,483,647	Double	Specialized	26.642	26.680	
			Generic	27.225	27.276	0.98

Table 6.4: Monte Carlo Integration Performance for TypeScript

Language	Test	Field	Variant	Real Time (s)	User Time (s)	Real Speedup
Java	1,000,000	Double	Specialized	0.054	0.049	
			Generic	0.061	0.047	0.89
	1,000,000,000	Double	Specialized	11.319	11.307	
			Generic	11.282	11.273	1.00
	2,147,483,647	Double	Specialized	24.199	24.188	
			Generic	24.047	24.031	1.01

6.2 Successive Over-Relaxation

Table 6.5: Successive Over-Relaxation Performance for Rust

Language	Test	Field	Variant	Real Time (s)	User Time (s)	Real Speedup
Rust	2000x2000	Double	Specialized	204.297	204.266	
			Generic	217.389	217.363	0.94
	3000x3000	Double	Specialized	482.967	482.909	
			Generic	500.026	499.943	0.97
	4000x4000	Double	Specialized	907.178	907.089	
			Generic	907.135	907.022	1.00

Table 6.6: Successive Over-Relaxation Performance for Java

Language	Test	Field	Variant	Real Time (s)	User Time (s)	Real Speedup
Java	2000x2000	Double	Specialized	181.911	181.924	
			Generic	385.587	536.068	0.47
	3000x3000	Double	Specialized	423.430	423.490	
			Generic	881.201	1297.404	0.48
	4000x4000	Double	Specialized	788.047	788.127	
			Generic	1747.981	4150.977	0.45

Table 6.7: Successive Over-Relaxation Performance for Go

Language	Test	Field	Variant	Real Time (s)	User Time (s)	Real Speedup
Go	2000x2000	Double	Specialized	183.076	183.457	
			Generic	305.981	306.522	0.60
	3000x3000	Double	Specialized	425.454	426.214	
			Generic	701.624	702.847	0.61
	4000x4000	Double	Specialized	792.201	793.612	
			Generic	1280.667	1282.907	0.62

Table 6.8: Successive Over-Relaxation Performance for TypeScript

Language	Test	Field	Variant	Real Time (s)	User Time (s)	Real Speedup
TypeScript	2000x2000	Double	Specialized	204.345	204.281	
			Generic	2655.779	7636.806	0.08
	3000x3000	Double	Specialized	471.462	471.408	
			Generic	6040.143	16 857.218	0.08
	4000x4000	Double	Specialized	877.230	877.189	
			Generic	11 654.490	32 125.131	0.08

6.3 LU Factorization

Table 6.9: LU Performance for Rust

Language	Test	Field	Variant	Real Time (s)	User Time (s)	Real Speedup
Rust	3000x3000	Double	Specialized	80.782	80.535	
			Generic	83.059	82.835	0.97
		Z mod p	Specialized	227.842	227.688	
			Generic	388.230	388.004	0.59
	4000x4000	Double	Specialized	211.823	211.336	
			Generic	206.415	206.014	1.03
		Z mod p	Specialized	541.558	541.296	
			Generic	923.085	922.676	0.59
	5000x5000	Double	Specialized	415.178	414.411	
			Generic	401.041	400.415	1.04
		Z mod p	Specialized	1058.023	1057.608	
			Generic	1813.817	1813.156	0.58

Table 6.10: LU Performance for Java

Language	Test	Field	Variant	Real Time (s)	User Time (s)	Real Speedup
Java	3000x3000	Double	Specialized	59.478	59.722	
			Generic	581.393	748.416	0.10
		Z mod p	Specialized	148.952	149.163	
			Generic	704.480	898.994	0.21
	4000x4000	Double	Specialized	149.205	149.588	
			Generic	1655.976	2190.007	0.09
		Z mod p	Specialized	352.086	352.345	
			Generic	1770.744	2330.948	0.20
	5000x5000	Double	Specialized	257.602	258.665	
			Generic	3311.219	4598.818	0.08
		Z mod p	Specialized	690.058	690.749	
			Generic	3449.471	4782.010	0.20

Table 6.11: LU Performance for Go

Language	Test	Field	Variant	Real Time (s)	User Time (s)	Real Speedup
Go	3000x3000	Double	Specialized	76.069	76.267	
			Generic	269.064	269.607	0.28
		Z mod p	Specialized	484.693	485.635	
			Generic	487.675	488.574	0.99
	4000x4000	Double	Specialized	187.097	187.486	
			Generic	646.843	648.134	0.29
		Z mod p	Specialized	1150.221	1152.405	
			Generic	1156.428	1158.593	0.99
	5000x5000	Double	Specialized	396.924	370.683	
			Generic	1254.800	1257.227	0.31
		Z mod p	Specialized	2243.925	2248.137	
			Generic	2254.344	2258.526	1.00

Table 6.12: LU Performance for TypeScript

Language	Test	Field	Variant	Real Time (s)	User Time (s)	Real Speedup
TypeScript	3000x3000	Double	Specialized	149.564	149.892	
			Generic	5041.809	15 547.949	0.03
		Z mod p	Specialized	1597.422	1597.786	
			Generic	7381.215	18 621.804	0.22
	4000x4000	Double	Specialized	355.874	356.549	
			Generic	17 010.358	50 678.162	0.02
		Z mod p	Specialized	3794.450	3795.169	
			Generic	21 570.547	56 748.553	0.18
	5000x5000	Double	Specialized	684.723	685.598	
			Generic	DNF	DNF	N/A
		Z mod p	Specialized	7402.994	7403.879	
			Generic	DNF	DNF	N/A

6.4 Fast Fourier Transform

Table 6.13: FFT Performance for Rust

Language	Test	Field	Variant	Real Time (s)	User Time (s)	Real Speedup
Rust	1,048,576	Complex	Specialized	1.700	1.694	
			Generic	1.844	1.837	0.92
		Z mod p	Specialized	4.238	4.229	
			Generic	4.188	4.180	1.01
	16,777,216	Complex	Specialized	63.830	63.795	
			Generic	68.877	68.282	0.93
		Z mod p	Specialized	102.473	102.322	
			Generic	110.602	110.337	0.93
	67,108,864	Complex	Specialized	312.835	312.701	
			Generic	332.258	330.317	0.94
		Z mod p	Specialized	484.428	483.882	
			Generic	506.262	505.274	0.96

Table 6.14: FFT Performance for Java

Language	Test	Field	Variant	Real Time (s)	User Time (s)	Real Speedup
Java	1,048,576	Complex	Specialized	1.937	1.958	
			Generic	9.504	16.253	0.20
		Z mod p	Specialized	4.430	4.499	
			Generic	8.689	10.659	0.51
	16,777,216	Complex	Specialized	72.732	72.742	
			Generic	495.413	2516.505	0.15
		Z mod p	Specialized	116.868	116.965	
			Generic	196.050	351.133	0.62
	67,108,864	Complex	Specialized	376.269	376.216	
			Generic	2666.021	20 163.663	0.14
		Z mod p	Specialized	558.103	557.784	
			Generic	1770.766	6301.280	0.32

Table 6.15: FFT Performance for Go

Language	Test	Field	Variant	Real Time (s)	User Time (s)	Real Speedup
Go	1,048,576	Complex	Specialized	1.903	1.899	
			Generic	4.615	4.621	0.41
		Z mod p	Specialized	4.417	4.423	
			Generic	8.432	8.441	0.52
	16,777,216	Complex	Specialized	70.848	71.043	
			Generic	132.091	132.286	0.54
		Z mod p	Specialized	113.829	114.004	
			Generic	171.090	171.411	0.67
	67,108,864	Complex	Specialized	354.122	355.064	
			Generic	661.326	662.650	0.54
		Z mod p	Specialized	513.923	514.771	
			Generic	771.498	772.337	0.67

Table 6.16: FFT Performance for TypeScript

Language	Test	Field	Variant	Real Time (s)	User Time (s)	Real Speedup
TypeScript	1,048,576	Complex	Specialized	2.243	2.195	
			Generic	79.950	170.886	0.03
		Z mod p	Specialized	12.730	12.580	
			Generic	44.122	70.114	0.29
	16,777,216	Complex	Specialized	89.966	89.844	
			Generic	2067.573	3729.290	0.04
		Z mod p	Specialized	299.634	298.580	
			Generic	1310.056	2329.277	0.23
	67,108,864	Complex	Specialized	377.381	377.103	
			Generic	23 424.406	44 634.464	0.016
		Z mod p	Specialized	1305.089	1301.358	
			Generic	6390.899	10 783.897	0.204

6.5 Naive Gröbner Basis

Table 6.17: Naive Gröbner Basis Performance for Rust

Language	Test	Exponent	Variant	Real Time (s)	User Time (s)	Real Speedup
Rust	Cyclic-4	Vector	Specialized	0.010	0.009	
			Generic	0.010	0.008	1.00
		Bitpacked	Specialized	0.004	0.001	
			Generic	0.005	0.003	0.80
	Cyclic-5	Vector	Specialized	1.890	1.888	
			Generic	1.960	1.957	0.96
		Bitpacked	Specialized	0.439	0.435	
			Generic	0.427	0.421	1.03
	Cyclic-6	Vector	Specialized	2495.891	2495.759	
			Generic	2906.039	2905.791	0.86
		Bitpacked	Specialized	371.081	371.059	
			Generic	384.535	384.512	0.97

Table 6.18: Naive Gröbner Basis Performance for Java

Language	Test	Exponent	Variant	Real Time (s)	User Time (s)	Real Speedup
Java	Cyclic-4	Vector	Specialized	0.113	0.366	
			Generic	0.111	0.306	1.02
		Bitpacked	Specialized	0.094	0.262	
			Generic	0.075	0.155	1.25
	Cyclic-5	Vector	Specialized	3.323	4.190	
			Generic	3.323	4.321	1.00
		Bitpacked	Specialized	1.145	1.727	
			Generic	1.305	1.818	0.88
	Cyclic-6	Vector	Specialized	5402.354	5406.971	
			Generic	8099.107	8105.132	0.67
		Bitpacked	Specialized	920.250	922.898	
			Generic	1547.445	1551.721	0.59

Table 6.19: Naive Gröbner Basis Performance for Go

Language	Test	Exponent	Variant	Real Time (s)	User Time (s)	Real Speedup
Go	Cyclic-4	Vector	Specialized	0.018	0.010	
			Generic	0.021	0.012	0.86
		Bitpacked	Specialized	0.012	0.004	
			Generic	0.016	0.010	0.75
	Cyclic-5	Vector	Specialized	4.694	4.998	
			Generic	5.093	5.301	0.92
		Bitpacked	Specialized	2.417	2.543	
			Generic	2.939	3.024	0.82
	Cyclic-6	Vector	Specialized	6208.807	6629.856	
			Generic	8786.183	9221.363	0.71
		Bitpacked	Specialized	3007.487	3052.588	
			Generic	4648.883	4682.216	0.65

Table 6.20: Naive Gröbner Basis Performance for TypeScript

Language	Test	Exponent	Variant	Real Time (s)	User Time (s)	Real Speedup
TypeScript	Cyclic-4	Vector	Specialized	0.067	0.074	
			Generic	0.084	0.090	0.80
		Bitpacked	Specialized	0.079	0.082	
			Generic	0.081	0.086	0.98
	Cyclic-5	Vector	Specialized	6.440	6.686	
			Generic	5.835	5.982	1.10
		Bitpacked	Specialized	4.778	4.965	
			Generic	8.143	8.197	0.59
	Cyclic-6	Vector	Specialized	10 764.579	10 762.419	
			Generic	11 902.993	11 902.308	0.90
		Bitpacked	Specialized	4528.744	4528.084	
			Generic	8238.359	8234.409	0.55

Table 6.21: Naive Gröbner Basis Performance for C++

Language	Test	Exponent	Variant	Real Time (s)	User Time (s)	Real Speedup
C++	Cyclic-4	Vector	Specialized	0.005	0.004	
			Generic	0.008	0.008	0.63
		Bitpacked	Specialized	0.003	0.002	
			Generic	0.003	0.003	1.00
	Cyclic-5	Vector	Specialized	2.915	2.912	
			Generic	5.203	5.200	0.56
		Bitpacked	Specialized	0.602	0.602	
			Generic	0.637	0.633	0.95
	Cyclic-6	Vector	Specialized	4031.537	4031.331	
			Generic	4879.393	4879.257	0.83
		Bitpacked	Specialized	553.886	553.865	
			Generic	587.880	587.862	0.94

Table 6.22: Naive Gröbner Basis Performance for Julia

Language	Test	Exponent	Variant	Real Time (s)	User Time (s)	Real Speedup
Julia	Cyclic-4	Vector	Specialized	1.708	2.730	
			Generic	2.574	3.567	0.66
		Bitpacked	Specialized	1.424	2.444	
			Generic	2.376	3.360	0.60
	Cyclic-5	Vector	Specialized	17.247	18.257	
			Generic	21.185	22.138	0.81
		Bitpacked	Specialized	7.500	8.526	
			Generic	8.827	9.826	0.85
	Cyclic-6	Vector	Specialized	12 600.175	12 588.242	
			Generic	14 932.440	14 922.942	0.84
		Bitpacked	Specialized	4862.552	4855.845	
			Generic	5235.569	5232.625	0.93

6.6 Output Artifact Sizes

The following section discusses how the compilation strategies used by each platform influence output artifact sizes. However, direct comparisons require several caveats, since output size depends on numerous factors including runtime packaging, standard library inclusion, metadata generation, linkage configuration, and compilation strategy. For example, Java bytecode artifacts and native executables rely on different categories of external runtime infrastructure, meaning that artifact size should not be interpreted as a directly equivalent measure of deployment footprint.
As a representative sample across platforms, only the three Gröbner basis implementations were compared directly. For platforms implementing the complete benchmark suite, the size of the fully compiled suite was also measured.
The results are summarized in the following table.

Table 6.23: Compiled Binary Sizes Across Languages and File Sets

Platform	**Z mod p/Vec exps**	**Z mod p/Bit exps**	**Gen/Gen exps**	**Full Test Suite**
Rust	527.58 KB	496.96 KB	572.73 KB	836.73 KB
Java	20.02 KB	17.87 KB	49.75 KB	159.37 KB
Go	2281.54 KB	2297.46 KB	2365.33 KB	2607.96 KB
TypeScript	18.51 KB	10.75 KB	39.71 KB	121.86 KB
C++	65.25 KB	57.24 KB	115.87 KB	
Julia	12.75 KB	15.08 KB	27.95 KB	

Chapter 7

Analysis

7.1 Overview

Design Constraints

Although methodological consistency was maintained where possible during development for each platform, there are naturally constraints specific to each platform that may influence the observed results. These constraints may prevent a benchmark from being implemented identically across platforms or require implementation decisions that influence performance independently of the generic programming model itself.

Results

The results are analyzed by each algorithm, in the same order as they were reported. Each discussion begins with observations that may apply across platforms, followed by platform-specific discussion where relevant.

Development Experience

An important aspect of developing with generics is the development experience of working with these platforms. Although an individual's selection of platform may be influenced primarily by the performance of each platform, the feasibility of reaching that potential performance requires the ability to develop the code. In real-world development, factors such as maintainability, ergonomics, tooling support, and implementation complexity can significantly influence language selection. Accordingly, this subsection discusses the practical development experience associated with each platform, including factors that improved or hindered implementation.

7.2 Design Constraints

Although each benchmark was implemented with the goal of maintaining methodological consistency across platforms, several language- and platform-specific constraints influenced implementation decisions. These constraints must be considered when interpreting the benchmark results, as they may introduce performance characteristics unrelated to the generic programming model itself.

TypeScript

TypeScript provides only two primitive numeric types: *number* and *BigInt*. The *number* type is the standard numeric representation in the language and corresponds to a 64-bit IEEE-754 floating-point value, consistent with JavaScript. This representation is available across all studied platforms in some form. In contrast, *BigInt* is TypeScript's only integer-specific primitive and represents arbitrarily large integers.

This limitation becomes relevant when an integer numeric type is required, particularly for finite-field arithmetic and bitpacked integer representations. For bitpacked exponent vectors, *BigInt* was selected due to the need for explicit bitwise manipulation and custom interpretation of bitstrings. That said, because *BigInt* is an arbitrary-precision type rather than a fixed-width primitive, its use is expected to introduce performance overhead relative to the fixed-size integer types available in other platforms.

The choice of representation for finite fields is less straightforward. In this study, the *number* type was used for finite-field implementations in order to preserve the naïve design philosophy adopted throughout the benchmark suite, as *number* functions as the default general-purpose numeric type in TypeScript. Furthermore, IEEE-754 double-precision values can represent all integers exactly up to 2^{53}, which is sufficient for the finite-field sizes considered in this study. This decision is also motivated by the relatively recent introduction of *BigInt*, which was added in TypeScript 3.2 in 2018.

Java

A practical limitation encountered during FFT benchmark development concerned the maximum test size supported by Java. Unlike some other studied platforms, Java arrays are indexed using the *int* type and cannot exceed the maximum signed 32-bit integer value of $2,147,483,647$ elements.

This limitation became relevant in the FFT benchmark due to the use of a prime sieve to determine congruent primes corresponding to test sizes. For an FFT input size of $268,435,456$, the required prime modulus was $3,221,225,473$, exceeding the representable

range of a signed 32-bit integer and preventing allocation of the corresponding sieve structure. Consequently, the maximum FFT input size for Java was reduced to $67,108,864$, the next smaller tested power of two.

Rust

Rust requires the size of fixed-length arrays to be known at compile time. As a result, fixed-size stack-allocated arrays cannot be used for benchmark inputs whose dimensions are determined dynamically at runtime. Instead, dynamically sized vectors must be used for parameterized data structures.
This constraint is particularly relevant for matrix-based benchmarks such as Successive Over-Relaxation and LU Factorization, where large numbers of repeated memory accesses occur. The requirement to use dynamically sized vectors results in a different data representation and allocation model than would be possible with fixed-size arrays. These differences may affect memory locality, allocation behaviour, and compiler optimization opportunities, which could influence observed performance. However, the extent to which any individual factor contributes to the measured results cannot be determined without further profiling.

Gröbner Basis

As a symbolic computation benchmark, Gröbner basis computation requires exact arithmetic and restricts coefficient representations to domains that can be represented exactly in software, thus excluding floating-point arithmetic.
An additional implementation challenge arises when considering rational coefficients. Both the findings of Arnold and the investigations conducted in this study indicate that coefficient swell presents a significant obstacle when applying Buchberger's algorithm over the rationals. As Arnold notes, "Intermediate coefficient swell is a well-known difficulty with Buchberger's algorithm for computing Gröbner bases over the rational numbers." Due to the substantial precision requirements imposed by coefficient swell, rational arithmetic was excluded from the benchmark implementation in favour of finite-field coefficient representations.
A further limitation concerns the bitpacked exponent vector representation used in the optimized monomial implementation. The 64-bit packed representation allocates space for six exponents at eight bits each and one total degree field of sixteen bits. Consequently, this representation supports at most six variables, restricting the largest cyclic benchmark instance that can be evaluated using the bitpacked form to cyclic 6. Larger cyclic systems would require more exponent slots than the representation permits.

7.3 Results

The following section presents the results and discusses any interesting observations, including potential explanations for outcomes that deviate from expectations. It begins with a subsection outlining the general timing expectations, and then proceeds to examine the results of each algorithm in order.

7.3.1 Expectations

Prior to examining the benchmark results, it is useful to establish expected performance trends in order to provide a baseline against which observed deviations may be interpreted. Although the evaluated languages differ substantially in syntax, runtime environment, and type systems, many of the observed performance characteristics can be understood in terms of when and how generic code is specialized. Languages such as Rust and C++ primarily rely on ahead-of-time monomorphization, generating specialized implementations for concrete types at compile time. Java and TypeScript instead rely more heavily on erased or uniform runtime representations, often introducing additional indirection and reducing opportunities for ahead-of-time specialization. Go occupies a hybrid position through shape-based compilation and dictionary mechanisms, while Julia performs aggressive specialization dynamically through just-in-time compilation. These differing approaches provide a useful framework for interpreting the performance trade-offs observed throughout the benchmarks.

Although generic realization strategy provides a useful framework for interpreting performance trends, the observed results are also influenced by many other factors, including memory layout, allocation behaviour, cache locality, compiler optimization success, runtime system behaviour, and implementation-specific design decisions. Consequently, the discussions that follow should be interpreted as plausible explanations of observed behaviour rather than definitive causal attribution.

Rust

Rust employs monomorphization for generic code generation, producing specialized machine code for each instantiated generic type at compile time. Consequently, the generic implementations are expected to perform only marginally slower than their specialized counterparts, with any overhead arising primarily from the abstraction of arithmetic operations and the inability to exploit benchmark-specific optimizations available in hand-specialized implementations. Given Rust's emphasis on systems-level performance and compile-time specialization, generic implementations are expected to approach the perfor-

mance of specialized implementations when the compiler is able to optimize abstractions effectively.

Java

Java implements generics through type erasure, replacing generic type parameters with their upper bounds during compilation. As a result, generic implementations are expected to incur measurable overhead relative to specialized versions due to factors such as boxing, object indirection, type casting, and reduced opportunities for ahead-of-time specialization where these occur. Accordingly, Java's generic benchmarks are expected to perform noticeably slower than their specialized equivalents.

Go

Go's generic implementation strategy incorporates elements resembling both monomorphization and dictionary-based approaches depending on usage context and compiler optimization opportunities. Consequently, generic implementations are expected to exhibit modest overhead relative to specialized implementations, with performance degradation anticipated to fall between the near-zero overhead of monomorphic systems and the larger penalties associated with runtime-dispatched approaches.

TypeScript

Because TypeScript generics exist only at compile time and are erased during transpilation to JavaScript, the generic type system itself imposes no direct runtime overhead. However, the transition from specialized numeric implementations to generic abstractions requires replacing primitive numeric values with wrapper objects representing field elements. This design change is expected to introduce substantial runtime overhead through increased object allocation, indirection, and method dispatch. While the precise magnitude of this slowdown is difficult to predict, generic implementations are expected to perform markedly worse than specialized versions.

C++

C++ utilizes template-based monomorphization for generic programming, generating specialized code for each instantiated type at compile time. As with Rust, only minimal performance degradation is expected when transitioning from specialized to generic implementations. Given C++'s mature optimization toolchain and longstanding reputation

for high-performance systems programming, it is expected to rank among the strongest performers across all benchmarks.

Julia

Julia specializes methods on concrete argument types at runtime through just-in-time compilation, often yielding execution behaviour similar to monomorphic specialization once compilation and type inference have occurred successfully. Consequently, steady-state execution performance is expected to exhibit minimal overhead when comparing specialized and generic implementations. However, because compilation occurs during execution, benchmark timings that include compilation overhead may reflect additional runtime cost proportional to the number of instantiated generic specializations. Despite this, Julia's design emphasis on generic scientific computing suggests that competitive performance should still be observed.

Finite Fields

Independent of platform-specific generic implementation strategy, finite-field benchmarks are expected to incur additional overhead when transitioning from specialized to generic implementations due to the abstraction of modular arithmetic operations. In specialized implementations, modulus reduction may be deferred or applied selectively when mathematically safe to do so, reducing the total number of modulus operations performed. In contrast, generic finite-field abstractions typically enforce reduction after every arithmetic operation in order to preserve correctness and encapsulation. This increased frequency of modulus operations is expected to contribute additional overhead in generic finite-field benchmarks across all platforms.

7.3.2 Monte Carlo Integration

With linear time complexity $O(n)$ and minimal memory or control-flow complexity, the Monte Carlo integration benchmark serves as a baseline for evaluating the raw overhead introduced by generic abstraction in a computationally simple workload. Because the algorithm consists primarily of repeated arithmetic operations with limited data structure manipulation, performance differences observed in this benchmark are expected to reflect abstraction overhead more directly than algorithm-specific implementation effects.
Across all platforms, the transition from specialized to generic implementations produced the smallest relative performance losses in this benchmark compared to the remainder of the suite. This outcome is consistent with expectations, as the simplicity of the algorithm

minimizes opportunities for abstraction overhead to compound through memory allocation, complex control flow, or nested arithmetic expressions. Consequently, the Monte Carlo benchmark establishes a baseline trend against which the more computationally intensive benchmarks may be compared.

7.3.3 Successive Over-Relaxation

The Successive Over-Relaxation (SOR) benchmark performs a single relaxation sweep over a two-dimensional grid, yielding time complexity of $O(n^2)$. Unlike Monte Carlo integration, SOR places greater emphasis on memory access patterns and repeated array indexing, making it more sensitive to differences in memory representation and data structure implementation across platforms.

Rust

Contrary to expectations, Rust exhibited the slowest specialized SOR performance among the studied platforms. One likely contributor is the implementation itself, which performs row cloning operations within the sweep loop in order to satisfy borrowing constraints associated with the chosen `Vec<Vec<f64>>` representation and in-place updates. Because these cloning operations occur repeatedly during iteration, they may introduce additional allocation and copying overhead directly within the benchmark's critical execution path.
Rust's requirement that fixed-size arrays have compile-time known dimensions necessitated the use of a dynamically allocated matrix representation for this benchmark. Combined with the row cloning performed during iteration, this introduces additional allocation and copying behaviour that may influence performance. However, the precise contribution of the underlying vector representation relative to these implementation choices cannot be determined without further low-level profiling and analysis.
Despite this lower-than-expected baseline performance, the transition from specialized to generic implementations in Rust produced only negligible slowdown, consistent with expectations for a monomorphized generic system.

Java

Whereas Java exhibited only modest overhead in Monte Carlo integration, the performance loss observed in SOR when moving from specialized to generic implementations is substantially larger. Given the increased arithmetic density and frequency of array accesses in SOR, one plausible explanation is that the overhead associated with boxed numeric objects and indirect method dispatch compounds significantly more in this benchmark than

in the simpler Monte Carlo case. However, detailed profiling would be required to confirm the precise cause.

Go

Go demonstrates performance degradation greater than Rust but smaller than Java when transitioning from specialized to generic implementations. This result is broadly consistent with expectations given Go's hybrid generic implementation strategy, suggesting that its abstraction overhead remains moderate even in a memory-intensive workload such as SOR.

TypeScript

TypeScript exhibits extremely large performance degradation when transitioning from specialized to generic implementations, with the generic SOR benchmark requiring approximately thirteen times longer to execute. This substantial slowdown likely reflects the combined cost of repeated object allocation, property access indirection, dynamic dispatch, and the replacement of primitive arithmetic with object-based abstractions. The scale of the slowdown suggests that the object-based numeric abstractions used in this implementation appear particularly costly for high-frequency memory- and arithmetic-intensive workloads within the JavaScript runtime environment.

7.3.4 LU Factorization

LU factorization shares many of the characteristics observed in the SOR benchmark, but with increased computational intensity due to its $O(n^3)$ time complexity. As a result, any overhead introduced by abstraction, memory access patterns, or arithmetic operations is amplified, leading to more pronounced performance differences between specialized and generic implementations.
In addition to floating-point arithmetic, this benchmark introduces finite-field implementations. A key distinction between these representations is that division operations in floating-point arithmetic are replaced by modular inverses in finite fields. Computing a modular inverse is significantly more expensive than a standard floating-point division, and as such, finite-field implementations are expected to exhibit slower performance across all platforms.

Rust

Rust demonstrates strong performance overall, with generic floating-point implementations slightly outperforming their specialized counterparts in some cases. Although the

magnitude of this improvement is small, it suggests that compiler optimizations or minor implementation differences may outweigh the expected abstraction overhead. This behaviour is consistent with Rust's monomorphization strategy and its ability to aggressively optimize generic code.
Nevertheless, in comparison to other platforms such as Go and Java, Rust's overall performance remains lower than expected. As noted previously, this may reflect implementation-specific overheads related to allocation, copying, and memory access behaviour associated with the chosen matrix representation, rather than generic programming overhead itself. Further profiling would be required to determine the relative importance of these factors.
For finite-field implementations, the transition from specialized to generic code results in performance degradation, in contrast to the floating-point case. This behaviour is consistent with expectations, as the abstraction of arithmetic operations in the generic implementation enforces more frequent modulus reductions, increasing computational cost. Additional contributing factors may be present, though further profiling would be required to identify them.

Java

Java exhibits a larger performance gap between specialized and generic implementations in LU factorization than was observed in SOR. This is consistent with the increased computational complexity of the algorithm, which amplifies the overhead associated with object boxing, method dispatch, and type erasure. As arithmetic operations and memory accesses occur more frequently, the cumulative cost of these abstractions becomes more pronounced.

Go

Go presents an asymmetry between floating-point and finite-field implementations. While the floating-point benchmarks show performance degradation consistent with or exceeding that observed in SOR, the finite-field implementations exhibit only minimal additional overhead when transitioning to generic code. This behaviour deviates from expectations and suggests that the interaction between Go's hybrid generic implementation model and the structure of the finite-field abstractions may differ from that of the floating-point case. Further investigation would be required to determine the underlying cause.

TypeScript

TypeScript exhibits substantial performance degradation in LU factorization when transitioning from specialized to generic implementations. In several cases, execution times

increase by more than an order of magnitude, with some larger test cases failing to complete within practical time limits. This trend is consistent with, but more pronounced than, the behaviour observed in the SOR benchmark, reflecting the increased computational intensity of LU factorization.
The observed slowdown is plausibly related to the cumulative cost of object allocation, dynamic dispatch, and the replacement of primitive arithmetic operations with object-based abstractions. Additionally, finite-field implementations incur further overhead due to the reliance on modulus operations and the absence of efficient fixed-width integer primitives in the JavaScript runtime. These factors likely contribute to the observed poor performance.

7.3.5 Fast Fourier Transform

The Fast Fourier Transform (FFT) benchmark differs from the preceding tests in that it introduces an explicit complex field abstraction, in addition to supporting finite-field arithmetic. Unlike LU factorization, FFT involves relatively few division or modular inverse operations. Despite this, finite-field implementations are still expected to perform more slowly than their floating-point counterparts due to the increased cost of modulus operations and the lack of hardware-level optimization for integer modular arithmetic.
A key characteristic of this implementation is that complex arithmetic is expressed through object-based abstractions. As a result, operations on complex numbers may incur significant overhead due to the creation of intermediate objects, particularly in generic implementations where arithmetic is further abstracted through base field operations.

Rust

Rust exhibits stable performance across both floating-point and finite-field FFT implementations, consistent with trends observed in earlier benchmarks. The relative difference between specialized and generic implementations remains small, suggesting that Rust's monomorphization strategy effectively minimizes abstraction overhead even in the presence of complex arithmetic. Additionally, Rust appears less sensitive to the object allocation overhead that impacts other platforms, potentially reflecting successful compiler optimization of temporary values and intermediate operations.

Java

Java demonstrates an inversion in relative performance between floating-point and finite-field implementations when comparing specialized and generic FFTs. In the specialized

case, the floating-point implementation performs faster than the finite-field version, as expected. However, in the generic implementation, the finite-field version outperforms the floating-point version.
One plausible explanation for this behaviour is the overhead associated with object allocation in complex arithmetic. In this implementation, each complex multiplication involves multiple base field operations (four multiplications, one addition, and one subtraction), each producing intermediate objects, followed by the construction of a final complex object. Consequently, a single complex multiplication may result in several heap allocations. In floating-point implementations, where complex numbers are represented using object wrappers around primitive values, this allocation cost becomes significant when compounded across the many operations performed in FFT.
In contrast, finite-field implementations may exhibit relatively lower allocation overhead depending on their internal representation and operation structure, leading to the observed inversion. This explanation is consistent with the allocation-heavy nature of the benchmark, though detailed profiling would be required to confirm the precise mechanisms.

Go

Go exhibits behaviour similar to Java, though less pronounced. The relative performance gap between floating-point and finite-field implementations narrows when transitioning from specialized to generic code, indicating that floating-point complex arithmetic incurs additional overhead under abstraction. For example, at $n = 67{,}108{,}864$, the relative performance difference between floating-point and finite-field implementations decreases when moving to generic code. This trend suggests that, as in Java, object allocation and abstraction overhead disproportionately affect complex floating-point operations.

TypeScript

TypeScript demonstrates the same qualitative behaviour observed in Java and Go, but at a significantly larger scale. The overhead associated with object-based arithmetic, dynamic dispatch, and repeated allocation of intermediate objects results in substantial performance degradation in generic implementations. As in previous benchmarks, the transition from specialized to generic code leads to large slowdowns, which are further amplified in the FFT due to the high frequency of arithmetic operations and intermediate object creation inherent to the algorithm.

7.3.6 Naive Gröbner Basis

The Gröbner basis benchmark represents the main symbolic computation workload in this test suite and differs fundamentally from the preceding numerical benchmarks. Because all computations are performed over finite fields and exponent vectors consist solely of non-negative integers, differences in floating-point performance are eliminated. Instead, performance is primarily determined by each platform's handling of modular arithmetic and the efficiency of exponent vector representations.
Two representations of exponent vectors are considered: a standard array/vector-based representation and a bitpacked representation using fixed-width integer encoding. The latter is expected to provide significant performance benefits due to reduced memory usage and improved cache locality, as well as the ability to leverage efficient bitwise operations.
This benchmark also introduces C++ and Julia, extending the analysis to additional generic programming models. Expectations for these languages follow those outlined previously.

Rust

Rust exhibits behaviour consistent with expectations for a monomorphized generic system. Generic implementations remain close to their specialized counterparts, indicating that abstraction overhead remains relatively small even within this symbolic computation workload.

Java

Java continues to exhibit greater separation between specialized and generic implementations than the monomorphic systems. This behaviour is consistent with expectations for an erased generic model and mirrors trends observed in the numerical benchmarks.

Go

Go generally follows the trends observed in earlier benchmarks for both Rust and Java, with no major deviations in overall performance. However, when comparing exponent vector representations, the performance improvement achieved by the bitpacked implementation is notably smaller than expected.While Rust and Java exhibit substantially larger speedups, reaching approximately $6\times$ on the largest benchmark instances, Go achieves only approximately a $2\times$ improvement.
This reduced benefit suggests that the implementation and runtime characteristics of Go in this benchmark may limit the performance gains typically associated with bitpacked

representations. Alternatively, the overhead associated with abstraction in Go's generic implementation may offset some of the gains provided by the more compact representation. Further investigation would be required to determine the precise cause.

TypeScript

TypeScript exhibits behaviour similar to Go, with relatively modest performance improvements when transitioning from vector-based to bitpacked representations, typically in the range of 1.5–2.5×. One contributing factor is the use of *BigInt* for bitpacked arithmetic, which, as an arbitrary-precision type, lacks the efficiency of fixed-width integer operations available in other platforms.
As a result, the expected advantages of bit-level manipulation and compact representation are diminished. This suggests that, in the TypeScript/JavaScript runtime, the cost of arbitrary-precision arithmetic may outweigh the benefits typically associated with bitpacking.

C++

C++ demonstrates performance characteristics broadly similar to Rust, consistent with its use of template-based monomorphization for generic programming. The transition from specialized to generic implementations incurs only minimal overhead, indicating effective compile-time specialization.
Despite this, overall performance is somewhat lower than that observed in Rust. This may reflect differences in implementation details rather than inherent language limitations. In particular, achieving optimal performance in C++ often requires more explicit control over memory layout, inlining, and low-level optimizations. The relatively straightforward implementation used in this study may not fully exploit the performance potential of the language.

Julia

Julia exhibits minimal performance degradation when transitioning from specialized to generic implementations, consistent with its type-specializing just-in-time compilation model. In contrast, baseline performance is significantly lower than expected relative to other platforms.
One possible explanation is that the naive implementation used in this benchmark results in frequent memory allocation, placing increased pressure on Julia's garbage collector. In allocation-heavy symbolic computations such as Gröbner basis construction, this overhead may become a dominant factor. While this hypothesis is consistent with the observed

behaviour, further profiling would be required to confirm the extent to which memory management contributes to the performance gap.

7.3.7 Output Artifact Sizes

The output artifact sizes align closely with expectations based on each platform's compilation and distribution model. However, because artifact size depends on many factors, including compiler settings, linking strategies, and runtime dependencies, these results should be interpreted with appropriate caution.

To provide a consistent basis for comparison, artifact sizes were measured using the three Gröbner basis implementations across all platforms. For platforms where the full benchmark suite was implemented, the size of the complete compiled test suite is also reported. The results are summarized in Table 6.23.

Platforms that distribute source-level artifacts, such as Julia and TypeScript, produce the smallest outputs, as these consist primarily of human-readable code or lightweight transpiled representations rather than fully compiled machine code. In contrast, ahead-of-time compiled languages produce significantly larger artifacts due to the inclusion of compiled instructions and, in some cases, embedded runtime components.

Among ahead-of-time compiled platforms, dynamically linked binaries, such as those typically produced by Java and C++, are substantially smaller than statically linked equivalents. This is evident in Table 6.23, where Java artifacts remain below 50 KB for individual implementations, and C++ binaries remain comparatively compact. These smaller sizes reflect the reliance on external shared libraries and/or runtime environments.

Conversely, Go produces significantly larger binaries, exceeding 2 MB even for individual Gröbner basis implementations. This is consistent with Go's default compilation strategy, which produces statically linked executables and embeds much of the runtime directly into the binary. Rust also produces relatively large binaries compared to Java and C++, though smaller than those of Go, reflecting a balance between static linking and compiler optimizations.

Across the ahead-of-time compiled monomorphic platforms (Rust, C++, and Go), the increase in artifact size when transitioning from specialized to generic implementations is relatively consistent in absolute terms. This suggests that, although monomorphization generates additional code for each instantiated type, the resulting growth in binary size remains moderate within the scope of the studied benchmarks.

7.4 Development Experience

While performance and output artifact size provide quantitative measures of platform characteristics, they do not fully capture the practical considerations involved in implementing generic scientific software. In real-world scenarios, transitioning from specialized implementations to generic ones requires not only performance awareness but also consideration of developer productivity, language ergonomics, and tooling support. As such, this section provides a qualitative discussion of the development experience across the studied platforms, focusing on features relevant to implementing generic numerical and symbolic algorithms.

Rust

Rust's strict ownership model and strong static type system provide robust compile-time guarantees, resulting in excellent linting and error detection. This significantly reduces the likelihood of type-related errors and unintended data mutations. At the same time, this strictness comes at the cost of certain conveniences, such as function overloading.
For example, in the original SciGMark implementation, the *coerce* function is overloaded to support both value extraction and construction depending on the input signature. In Rust, this behaviour must be expressed through distinct functions with explicit names, such as *coerce*, *coerce_to_f64*, and *coerce_from_int*. While this approach improves clarity and type safety, it can increase verbosity in situations where overloading would otherwise be used.
Despite this, Rust provides several ergonomic features that improve code clarity in practice. Iterator utilities such as *.zip* enable concise traversal of multiple data structures, and operations such as *.swap* simplify in-place manipulation of arrays. These features are particularly beneficial in array-heavy numerical algorithms.
Additionally, Rust's trait system provides strong support for generic programming. Traits such as *Hash*, *Eq*, and *Clone* integrate seamlessly with the standard library, enabling functionality such as hashing, sorting, and deep copying with minimal additional implementation effort.

Java

Java provides a relatively straightforward development experience and serves as a baseline among the studied platforms. Its explicit conformance model for generics simplifies implementation, as required interfaces must be declared and implemented directly. The availability of function overloading further improves usability, allowing multiple behaviours

to be expressed under a single method name.
At the same time, Java lacks some of the convenience features available in other languages. For example, there is no direct equivalent to Rust's iterator combinators such as *.zip*, nor are there built-in utilities for operations such as array element swapping. While these limitations do not prevent implementation, they may result in more verbose or less expressive code in certain cases.

Go

Go's relatively recent introduction of generics results in a development experience that lacks some of the conveniences found in more mature systems. One limitation is the absence of a built-in, universal object equality mechanism. Unlike Rust, which supports equality through the == operator, or Java, which provides the *.equals* method, Go requires explicit implementation of equality logic where needed.
Similarly, Go does not provide a default hashing mechanism for user-defined types. Although hashing is supported, the developer must explicitly define the behaviour, increasing implementation effort relative to platforms with integrated hashing support.
Additionally, Go's use of implicit conformance for generics can reduce the effectiveness of static analysis tools. Because interface conformance is implicit, certain missing-method issues may only become apparent once generic code is instantiated or exercised, potentially complicating debugging relative to explicitly declared systems.

TypeScript

TypeScript shares several of the limitations observed in Go, particularly in the areas of object equality and hashing. The language does not provide a standard hashing mechanism for user-defined types, requiring either external libraries or custom implementations. In this study, a dictionary-based workaround was employed, introducing additional complexity and potential performance overhead.
Furthermore, the limited set of primitive numeric types in TypeScript presents challenges for scientific computing. While the *number* type can exactly represent integers up to 2^{53}, the lack of fixed-width integer types and reliance on arbitrary-precision *BigInt* for integer operations reduces both control and performance relative to other platforms.
Despite these limitations, TypeScript offers several ergonomic features. Array manipulation is relatively flexible, with built-in support for operations such as element swapping and index-based traversal patterns. Additionally, while TypeScript employs an implicit conformance model, the optional *implements* keyword enables improved linting and early detection of missing methods, partially mitigating the drawbacks of structural typing.

C++ and Julia

Due to the reduced implementation scope of C++ and Julia within this study, extensive qualitative evaluation of their development ergonomics would be inappropriate. However, some observations can still be made regarding their generic programming models.
Both languages employ forms of implicit conformance through duck typing or template-style structural matching. While this offers flexibility, it also weakens early error detection systems, such as linters, relative to systems with explicit conformance. In C++, missing required methods are typically detected only during template instantiation at compile time. In Julia, errors may surface only at runtime when an unsupported method is invoked. In both cases, these characteristics can increase debugging difficulty compared with platforms that enforce stricter design patterns.

7.5 Commonality and Divergence in Generic Systems

Although the studied platforms vary substantially in syntax, runtime design, and compilation strategy, the results of this study suggest that modern generic systems share a common conceptual foundation. Across all evaluated languages, it was possible to express the benchmark suite using parametric abstractions over arithmetic domains and polynomial structures. In each case, algorithms such as LU factorization, Fast Fourier Transform, and Gröbner basis computation could be implemented independently of the concrete numeric representation through the use of generic interfaces or parameterized types. This suggests the existence of a broadly portable core of generic programming functionality capable of expressing a wide range of numerical and symbolic algorithms across diverse language ecosystems.
Despite this shared expressive capability, the realization strategies used by each platform strongly influence runtime behaviour, artifact size, and development ergonomics. One of the most significant distinctions concerns when and how generic code is specialized.
Rust and C++ primarily employ ahead-of-time monomorphization, generating specialized machine code for each instantiated type during compilation. This approach enables aggressive compiler optimization and often preserves concrete data representations throughout compilation and execution, usually resulting in performance characteristics that closely resemble manually specialized code. The benchmark results consistently reflected this behaviour, with both languages typically exhibiting only minimal performance degradation when transitioning from specialized to generic implementations.
In contrast, Java and TypeScript rely more heavily on erased or uniform runtime representations. Java implements generics through type erasure, while TypeScript erases generic

information entirely during transpilation to JavaScript. In both systems, abstraction over arithmetic domains frequently required boxed object representations and indirect method dispatch, introducing additional allocation and indirection overhead. These costs appeared particularly significant in arithmetic-intensive workloads such as LU factorization and FFT, where repeated object creation and dispatch occurred within tight computational loops.

Go's implementation strategy incorporates elements resembling shape-based compilation and dictionary mechanisms, producing behaviour that generally fell between monomorphic and erased systems. This implementation strategy exhibits moderate overhead while avoiding some of the more substantial penalties observed in Java and TypeScript.

Julia instead utilizes dynamic specialization through just-in-time compilation. Rather than generating specialized code ahead of execution, Julia specializes methods on concrete runtime argument types as they are encountered. This can allow type-stable generic code to approach near-monomorphic steady-state execution performance while retaining a highly dynamic programming model. However, this strategy introduces additional compilation overhead during execution and increases reliance on runtime type inference and memory management.

Another important distinction lies in constraint systems and interface specification. Rust and Java employ explicit conformance models through traits and interfaces respectively, requiring types to explicitly declare supported behaviour. In contrast, Go, TypeScript, C++, and Julia rely more heavily on structural or implicit conformance models, where compatibility is determined by the presence of required operations rather than explicit declarations. These differing approaches influence both usability and error reporting. Explicit systems generally provide stronger compile-time guarantees and clearer diagnostics, while implicit systems offer greater flexibility and reduced boilerplate at the cost of weaker early error detection.

The benchmark results further demonstrate that runtime representation is often as important as the generic mechanism itself. In numerical kernels where arithmetic could remain on primitive or stack-allocated representations, generic abstractions often incurred little overhead. That said, when abstraction required boxed representations, heap allocation, or indirect dispatch, performance degradation became substantially more pronounced. This was especially evident in Java and TypeScript, where object-oriented numeric abstractions likely introduced significant allocation pressure in arithmetic-intensive workloads. Conversely, monomorphic systems capable of preserving concrete representations throughout compilation generally maintained substantially stronger performance characteristics.

These observations suggest that generic programming systems are best understood not as a binary distinction between "generic" and "non-generic" languages, but rather as a design space spanning multiple interacting dimensions, including specialization strategy, runtime representation, compilation model, and constraint system design. While many

modern languages support similar forms of parametric abstraction at the source level, the mechanisms used to realize those abstractions vary significantly, leading to trade-offs between runtime performance, binary size, compilation cost, flexibility, and implementation complexity.

More broadly, the results suggest that a portable generic core for scientific and symbolic computing is feasible at the level of algorithmic abstraction. Across all studied platforms, the same high-level structures and generic decomposition strategies could be implemented with relatively minor conceptual variation. However, the study also demonstrates that portability of abstraction does not imply portability of performance. Achieving competitive performance frequently required awareness of platform-specific characteristics such as allocation behaviour, runtime specialization, memory layout, and arithmetic representation. Consequently, while generic scientific software can often be expressed in a largely platform-independent manner, efficient realization remains strongly dependent on the underlying language and runtime design.

Table 7.1: Generic realization strategies and primary sources of overhead

Language	**Generic Realization Strategy**	**Primary Sources of Overhead**
Rust	Ahead-of-time monomorphization using traits and generics	Additional abstraction layers, allocation behaviour, and optimizer limitations in some workloads
C++	Template-based ahead-of-time monomorphization	Template instantiation cost, binary growth, and dependence on implementation details and compiler optimization
Java	Type erasure with runtime JIT optimization and speculative specialization	Boxing/unboxing, allocation pressure, indirect dispatch, and uniform object representations
Go	Hybrid strategy combining shape-based compilation and dictionary mechanisms	Interface indirection, dictionary passing, and runtime representation costs
TypeScript	Compile-time type erasure to JavaScript	Object wrappers, dynamic dispatch, allocation overhead, and lack of efficient primitive generic representations
Julia	Runtime specialization through JIT compilation and multiple dispatch	Compilation latency, allocation behaviour, garbage collection, and inference sensitivity

Table 7.2: Observed behaviour and application suitability of studied platforms

Language	**Typical Observed Behaviour**	**Best-Fit Application Profile**
Rust	Generic implementations often approached specialized performance, with relatively small overhead across most benchmarks	Performance-sensitive numerical or systems software where predictable runtime behaviour and strong compile-time guarantees are important
C++	Generic implementations generally remained close to specialized performance, though results were sensitive to implementation structure	High-performance numerical and systems applications requiring low-level control and mature optimization tooling
Java	Generic implementations often exhibited moderate to substantial overhead in arithmetic-intensive workloads	Long-running managed applications where portability, ecosystem maturity, and runtime optimization are prioritized
Go	Generic implementations typically showed moderate overhead between monomorphic and erased systems	Cloud, infrastructure, and concurrent systems where simplicity and deployment characteristics are prioritized over maximal numerical performance
TypeScript	Generic numeric abstractions often incurred substantial overhead in arithmetic-heavy workloads	Applications prioritizing flexibility and interoperability rather than compute-intensive numerical kernels
Julia	Type-stable generic code often achieved competitive steady-state performance, though allocation-heavy workloads could degrade performance	Interactive scientific computing and exploratory numerical programming benefiting from dynamic specialization

Chapter 8

Conclusions and Future Work

This thesis investigated how different generic realization strategies influence the trade-offs between abstraction and performance across modern programming languages. These trade-offs were examined through differences in specialization strategy, compilation model, memory management, and type system design. To support this analysis, a suite of arithmetic-intensive numerical and symbolic algorithms was implemented and benchmarked in both specialized and generic forms, enabling comparison not only of runtime performance, but also of factors such as binary size, portability, and development characteristics.

8.1 Key Findings

Across the evaluated workloads, a major determinant of generic performance was not the presence of abstraction itself, but the stage at which specialization occurred. Languages employing compile-time specialization through monomorphization often achieved performance close to manually specialized implementations, often remaining very close to their specialized counterparts in arithmetic-intensive numerical kernels. In contrast, languages relying on boxed representations or runtime polymorphism often exhibited additional overhead associated with indirection, allocation, and reduced optimization opportunities. These findings suggest that the cost of abstraction is not inherent to generic programming, but instead depends on the realization strategy used by the language and compiler.

The results further demonstrate that no single realization strategy is universally optimal. Compile-time specialization provides highly efficient execution and predictable performance characteristics, particularly in arithmetic-intensive numerical kernels, but often increases compilation cost and binary size due to code duplication across concrete

types. In contrast, approaches based on type erasure or uniform runtime representations reduce duplication and simplify deployment, but introduce overheads ranging from modest slowdowns to multi-fold performance degradation depending on the degree of boxing and runtime dispatch involved. Runtime specialization approaches occupy an intermediate position, trading predictability for adaptive optimization and flexibility.

The impact of these trade-offs also varied across workload categories. Numerical kernels with regular control flow and stable data representations often benefited strongly from ahead-of-time specialization, allowing generic implementations to approach or match the performance of specialized code. Symbolic workloads, however, exhibited more mixed behaviour due to irregular memory access patterns, dynamic allocation, and more complex control flow, reducing the relative advantage of static specialization.

Beyond runtime performance, the study also highlighted secondary trade-offs related to binary size, portability, and development experience. Compile-time specialization strategies generally produced larger output artifacts, while runtime-based approaches avoided code duplication at the cost of larger runtime dependencies. Ease of development was similarly influenced by language design choices, particularly the distinction between explicit and implicit conformance mechanisms, which affect compile-time guarantees, tooling support, and maintainability.

Despite substantial differences in runtime systems, compilation models, and type systems, the evaluated languages shared a broadly common abstraction model based on parametric polymorphism and constrained generic interfaces. Across platforms, the benchmark algorithms could generally be expressed using similar generic structures and reusable abstractions. The primary source of divergence therefore arose not from the expressiveness of generic programming itself, but from differences in realization strategy, runtime representation, and optimization behaviour.

8.2 Practical Implications

For practitioners, these results suggest that generic abstractions can generally be considered effectively “free” when the language is capable of specializing code ahead of time while frequently preserving concrete data representations throughout execution. This was particularly evident in arithmetic-intensive numerical kernels, where compile-time specialization strategies frequently enabled generic implementations to remain close to the performance of manually specialized code. In contrast, abstraction costs became more significant when generic representations introduced boxing, heap allocation, or runtime dispatch within tight computational loops.

The suitability of a particular realization strategy depends strongly on the target ap-

plication and its priorities. Performance-critical numerical workloads benefit most from ahead-of-time specialization, predictable memory layouts, and aggressive compiler optimization. In contrast, reusable library code and abstraction-heavy applications may prioritize reduced compilation complexity, smaller binaries, implementation flexibility, or ease of deployment over absolute runtime performance. Long-running or highly dynamic systems may additionally benefit from adaptive optimization through just-in-time compilation, while symbolic and irregular workloads exhibit more mixed behaviour due to the increased influence of memory management and allocation patterns.

Binary size and deployment characteristics also represent important practical considerations. Compile-time specialization may increase code duplication across concrete instantiations, while statically linked runtimes can substantially increase executable size. Conversely, dynamically linked or runtime-managed systems often reduce artifact size at the expense of additional deployment dependencies and reduced portability.

Among the evaluated languages, Rust provides an example of a language combining ahead-of-time compilation, monomorphization, and explicit interface constraints within a modern systems-programming environment. These characteristics enabled performance comparable to C++ while also providing strong portability and compile-time guarantees. However, the broader results of this study suggest that no single language or realization strategy is universally optimal. Instead, the appropriate choice depends on the performance requirements, deployment constraints, and development priorities of the target application.

8.3 Limitations

This study has several limitations that should be considered when interpreting the results. The observed performance differences should not be interpreted as arising solely from generic realization strategy in isolation. In many cases, factors such as allocation behaviour, memory layout, cache locality, compiler optimization success, and implementation-specific design choices also likely contributed to the measured results. Consequently, the findings of this study are best interpreted as demonstrating correlations between realization strategies and observed behaviour within the evaluated implementations and workloads, rather than establishing universal causal relationships.

Furthermore, the evaluation is based on a limited set of numerical and symbolic kernels, which may not fully represent the diversity of real-world workloads.

Additionally, benchmarking methodology has inherent limitations. For platforms utilizing just-in-time compilation, assumptions were made regarding warm-up behaviour, and execution time was measured using Bash's *time* command without accounting for factors such as multi-threading or fine-grained performance counters.

Finally, the study considers a relatively small set of programming languages. Although the selection was intended to cover a broad range of design choices, other languages may provide additional insights.

8.4 Future Work

There are several directions in which this work can be extended to provide a deeper understanding of the trade-offs associated with generic programming. The inclusion of additional programming languages would broaden the scope of the study. In particular, languages such as Fortran, Aldor, or Swift may offer further insight into alternative approaches to generics, compilation strategies, and runtime design. Additionally, a more detailed performance analysis using profiling tools would help identify the precise causes of the observed behaviour. While this study establishes high-level trends, further investigation into factors such as dynamic dispatch overhead, inlining decisions, memory layout, and cache utilization would provide a more complete explanation of performance differences between generic and specialized implementations. Future work could explore the impact of generics in parallel and heterogeneous computing environments. In particular, examining how generic abstractions interact with GPU programming models, vectorization, and tiling strategies may reveal additional trade-offs not captured in this study. Furthermore, expanding the set of benchmark algorithms to include more diverse workloads, such as memory-bound or I/O-bound tasks, would improve the generality of the conclusions. This would help determine whether the trends observed in arithmetic-intensive kernels extend to other domains.

8.5 Closing Statement

Ultimately, this work demonstrates that abstraction and performance are not inherently opposed. The cost of generic programming is determined less by abstraction itself than by the strategy used to realize it. Different approaches to specialization produce different trade-offs in runtime performance, binary size, portability, and predictability, and these trade-offs must be evaluated in the context of the target application rather than through a single notion of optimality.

References

[1] TOP500, "The LINPACK benchmark," https://www.top500.org/project/linpack, 2025, accessed: 2026-04-14.

[2] NASA Advanced Supercomputing Division, "NAS parallel benchmarks," https://www.nas.nasa.gov/software/npb.html, 2026, accessed: 2026-04-14.

[3] Innovative Computing Laboratory, "HPC challenge benchmark suite," https://hpcchallenge.org/hpcc, 2022, accessed: 2026-04-14.

[4] R. K. Charania, "Exploring and benchmarking high performance & scientific computing using R, R's HPC packages and lower level compiled languages : A comparative study," *arXiv*, Apr 2019.

[5] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman, "Benchmarking Java against C and Fortran for scientific applications," in *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*. New York, NY, USA: ACM, June 2001, pp. 97–105.

[6] R. Pozo and R. Miller, "Home page of SciMark2 project," http://math.nist.gov/scimark2, accessed: 2026-04-14.

[7] L. Dragan and S. M. Watt, "Performance analysis of generics in scientific computing," in *Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2005, p. 8.

[8] R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock, "A comparative study of language support for generic programming," in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, vol. 38, no. 11. ACM, Nov. 2003, pp. 115–134.

[9] S. Ellis, S. Zhu, N. Yoshida, and L. Song, "Generic Go to Go: Dictionary-passing, monomorphisation, and hybrid," *Proceedings of ACM on Programming Languages*, vol. 6, no. OOPSLA2, Oct 2022.

[10] R. Fateman, "Comparing the speed of programs for sparse polynomial multiplication," *SIGSAM Bulletin*, vol. 37, no. 1, pp. 4–15, Mar 2003.

[11] J. van der Hoeven, "Relax, but don't be too lazy," *Journal of Symbolic Computation*, vol. 34, no. 6, pp. 479–542, Dec 2002.

[12] J. Dongarra, J. Gunnels, H. Bayraktar, A. Haidar, and D. Ernst, "Hardware trends impacting floating-point computations in scientific applications," *arXiv*, Nov 2024, arXiv:2411.12090.

[13] A. Turon, "Abstraction without overhead: traits in Rust," https://blog.rust-lang.org/2015/05/11/traits, 2015, accessed: 2026-04-14.

[14] LLVM Project, "LLVM language reference manual," https://llvm.org/docs/LangRef.html, n.d., accessed: 2026-04-14.

[15] S. Klabnik, C. Nichols, and C. Krycho, "The Rust programming language: Defining shared behavior with traits," https://doc.rust-lang.org/book/ch10-02-traits.html, n.d., accessed: 2026-04-14.

[16] Oracle, "Type erasure," https://docs.oracle.com/javase/tutorial/java/generics/erasure.html, n.d., accessed: 2026-04-14.

[17] ——, "Java language specification, Java SE 21 — chapter 4: Types, values, and variables," https://docs.oracle.com/javase/specs/jls/se21/html/jls-4.html, n.d., accessed: 2026-04-14.

[18] I. L. Taylor and R. Griesemer, "Type parameters proposal," https://go.googlesource.com/proposal/+/refs/heads/master/design/43651-type-parameters.md, 2021, accessed: 2026-04-14.

[19] The Go Project, "Frequently asked questions (FAQ)," https://go.dev/doc/faq, n.d., accessed: 2026-04-14.

[20] Microsoft, "Generics," https://www.typescriptlang.org/docs/handbook/2/generics.html, Apr 2026, accessed: 2026-04-14.

[21] C. Bruni, "Fast properties in V8," https://v8.dev/blog/fast-properties, 2017, accessed: 2026-04-14.

[22] V8 Project, "V8 documentation," https://v8.dev/docs, n.d., accessed: 2026-04-14.

[23] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, Jan 2017.

[24] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, Sept 2012.

[25] The Julia Project, "Performance tips," https://docs.julialang.org/en/v1/manual/performance-tips, n.d., version 1.12.6, Accessed: 2026-04-14.

[26] D. Vandevoorde, N. M. Josuttis, and D. Gregor, *C++ Templates: The Complete Guide*, 2nd ed. Boston, MA: Addison-Wesley, 2018.

[27] B. Stroustrup, *The C++ Programming Language*, 4th ed. Upper Saddle River, NJ: Addison-Wesley, 2013.

[28] The Rust Release Team, "Announcing Rust 1.92.0," https://blog.rust-lang.org/2025/12/11/Rust-1.92.0, Dec 2025, accessed: 2026-04-15.

[29] The Rust Project Developers, "The Rust programming language: Generics," https://doc.rust-lang.org/1.0.0/book/generics.html, 2015, accessed: 2026-04-15.

[30] The Rust Core Team, "Announcing Rust 1.26," https://blog.rust-lang.org/2018/05/10/Rust-1.26, May 2018, accessed: 2026-04-15.

[31] Rust Release Team, "Announcing Rust 1.51.0," https://blog.rust-lang.org/2021/03/25/Rust-1.51.0, Mar 2021, accessed: 2026-04-15.

[32] J. Huey, "Generic associated types to be stable in Rust 1.65," https://blog.rust-lang.org/2022/10/28/gats-stabilization, Oct 2022, accessed: 2026-04-15.

[33] OpenJDK Contributors, "JDK 25: Java SE 25 reference implementation overview," https://openjdk.org/projects/jdk/25, 2025, accessed: 2026-04-15.

[34] Oracle, "Java SE development kit 25.0.1 release notes," https://www.oracle.com/java/technologies/javase/25-0-1-relnotes.html, Oct 2025, accessed: 2026-04-15.

[35] ——, "J2SE 5.0 new features and enhancements," https://docs.oracle.com/javase/1.5.0/docs/relnotes/features.html, 2004, accessed: 2026-04-15.

[36] ——, "Java HotSpot virtual machine performance enhancements," https://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html, 2011, accessed: 2026-04-15.

[37] OpenJDK Contributors, "JDK 8 features," https://openjdk.org/projects/jdk8/features, 2014, accessed: 2026-04-15.

[38] The Go Authors, "Go 1.25 release notes," https://go.dev/doc/go1.25, Aug 2025, accessed: 2026-04-15.

[39] ——, "Go 1.25.5 release notes," https://go.dev/doc/devel/release#go1.25.5, Dec 2025, accessed: 2026-04-15.

[40] ——, "Go 1.18 release notes," https://go.dev/doc/go1.18, Mar 2022, accessed: 2026-04-15.

[41] ——, "Introducing Go," https://go.dev/blog/hello-world, Nov 2009, accessed: 2026-04-15.

[42] ——, "Go 1.20 release notes," https://go.dev/doc/go1.20, 2023, accessed: 2026-04-15.

[43] ——, "Go 1.21 release notes," https://go.dev/doc/go1.21, 2023, accessed: 2026-04-15.

[44] D. Rosenwasser, "Announcing TypeScript 5.9," https://devblogs.microsoft.com/typescript/announcing-typescript-5-9/, Aug 2025, accessed: 2026-04-15.

[45] TypeScript Team, "TypeScript on npm," https://www.npmjs.com/package/typescript, 2025, version 5.9.3 published October 2025. Accessed: 2026-04-15.

[46] Node.js Contributors, "Node.js v24.12.0 release archive," https://nodejs.org/en/download/archive/v24.12.0, 2025, first released May 6, 2025; last updated Dec 10, 2025. Accessed: 2026-04-15.

[47] A. Hejlsberg, D. Rosenwasser, and L. Stevenson, "Announcing TypeScript 1.0," https://devblogs.microsoft.com/typescript/announcing-typescript-1-0/, Apr 2014, accessed: 2026-04-15.

[48] D. Rosenwasser, "Announcing TypeScript 2.1," https://devblogs.microsoft.com/typescript/announcing-typescript-2-1-2, Dec 2016, accessed: 2026-04-15.

[49] ——, "Announcing TypeScript 2.8," https://devblogs.microsoft.com/typescript/announcing-typescript-2-8-2, Mar 2018, accessed: 2026-04-15.

[50] ——, "Announcing TypeScript 4.1," https://devblogs.microsoft.com/typescript/announcing-typescript-4-1, Nov 2020, accessed: 2026-04-15.

[51] cppreference contributors, "C++11," https://en.cppreference.com/w/cpp/11.html, n.d., accessed: 2026-04-15.

[52] ——, "C++14," https://en.cppreference.com/w/cpp/14.html, n.d., accessed: 2026-04-15.

[53] ——, "C++17," https://en.cppreference.com/w/cpp/17.html, n.d., accessed: 2026-04-15.

[54] Julia Developers, "Julia downloads: Older unmaintained releases," https://julialang.org/downloads/oldreleases, n.d., accessed: 2026-04-15.

[55] ——, "Announcing the release of Julia 1.0," https://julialang.org/blog/2018/08/one-point-zero, Aug 2018, accessed: 2026-04-15.