High-Performance Computational Genomics

by

Ariya Shajii

B.S., Boston University (2016)

S.M., Massachusetts Institute of Technology (2018)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2021

Author .....

Department of Electrical Engineering and Computer Science

August 23, 2021

Certified by .....

Bonnie Berger

Simons Professor of Mathematics

Professor of Electrical Engineering and Computer Science

Thesis Supervisor

Certified by .....

Saman Amarasinghe

Professor of Electrical Engineering and Computer Science

Thesis Supervisor

Accepted by .....

Leslie A. Kolodziejski

Professor of Electrical Engineering and Computer Science

Chair, Department Committee on Graduate Students

High-Performance Computational Genomics

by

Ariya Shajii

Submitted to the Department of Electrical Engineering and Computer Science
on August 23, 2021, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Next-generation sequencing data is growing at an unprecedented rate, leading to new revelations in biology, healthcare, and medicine. Many researchers use high-level programming languages to navigate and analyze this data, but as gigabytes grow to terabytes or even petabytes, high-level languages become prohibitive and impractical for performance reasons. This thesis introduces Seq, a Python-based, domain-specific language for bioinformatics and genomics that combines the power and usability of high-level languages like Python with the performance of low-level languages like C or C++. Seq allows for shorter, simpler code, is readily usable by a novice programmer, and obtains significant performance improvements over existing languages and frameworks. Seq is showcased and evaluated by implementing a range of standard, widely-used applications from all stages of the genomics analysis pipeline, including genomic index construction, data pre- and post-processing, read mapping and alignment, and haplotype phasing. We show that the Seq implementations are up to an order of magnitude faster than existing hand-optimized implementations, with just a fraction of the code. Seq's substantial performance gains are made possible by a host of novel genomics-specific compiler optimizations that are out of reach for general-purpose compilers, coupled with a static type system that avoids all of Python's runtime overhead and object metadata. By enabling researchers of all backgrounds to easily implement high-performance analysis tools, Seq aims to act as a catalyst for scientific discovery and innovation. Finally, we also generalize many of the principles used by Seq to create a domain-configurable compiler called Codon, which can be applied to other domains with similar results.

Thesis Supervisor: Bonnie Berger
Title: Simons Professor of Mathematics
Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to firstly thank my research advisors, Bonnie Berger and Saman Amarasinghe. I first began working with Prof. Berger as a junior in high school, nearly 8 years ago at the time of writing. Her continued support, guidance and mentorship over this period have without question shaped me as a scientist, researcher, and individual. I also began working with Prof. Amarasinghe after my master's work at MIT, producing the work presented in this thesis. His precise and methodical approach to research and engineering problems taught me how to approach seemingly insurmountable challenges, a skill I will cherish for the rest of my scientific career. Prof. Berger and Prof. Amarasinghe offered me complementary advising styles and often viewed research questions in unique ways, affording me the opportunity to think about many different perspectives and viewpoints. I am beyond grateful to have had the opportunity to learn from and work under them.

I would also like to thank my academic advisor, Manolis Kellis, for his invaluable help and support with navigating my time at MIT. I would further like to thank Ibrahim Numanagić, whom I began working with during his time as a postdoc at MIT, and continue to collaborate with today—this thesis would not have been possible without Ibrahim's guidance and collaboration. I would like to thank the many Berger and Amarasinghe lab group members that I have had the privilege of interacting with over the years, particularly Deniz Yorukoglu and Yun William Yu, who advised and mentored me from an early stage. I would like to additionally thank Patrice Macaluso and Mary McDavitt for their steadfast kindness and support, administratively and otherwise.

Last but certainly not least, I would like to thank my family for their unwavering support throughout this journey: my mother (Haleh), father (Ali), brother (Aram) and sister (Kimia).

Previous Publications of This Work

Parts of this thesis are based on the following works:

- "Seq: a high-performance language for bioinformatics," which appeared in OOPSLA (2019) with co-authors Ibrahim Numanagić, Riyadh Baghdadi, Bonnie Berger and Saman Amarasinghe [109].

- "A Python-based programming language for high-performance computational genomics," which appeared in Nature Biotechnology (2021) with co-authors Ibrahim Numanagić, Alexander T. Leighton, Haley Greenyer, Saman Amarasinghe and Bonnie Berger [111].

- Yet to be published work on Codon, a generalization of Seq to different domains, with co-authors Gabriel Ramirez, Jessica Ray, Haris Smajlović, Ibrahim Numanagić, Bonnie Berger and Saman Amarasinghe.

Note that I, Ariya Shajii, was first or co-first author in each of these works.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Big data" has become a topic of popular interest in recent years, both in terms of what new insights various big data sources can provide us, as well as the unique computational challenges they pose. Over the past several decades, fields like physics, cosmology, meteorology, finance, and indeed genomics, have all been revolutionized by our newfound ability to store and analyze massive datasets at scale. Even as data continued to increase exponentially in size, so too did our computational power thanks to the promise of Moore's Law [107]. However, for the first time since its inception, we are failing to meet the exponential trend posited by Gordon Moore in the mid-1960s. As a result, high-performance algorithms, methods, and tools are now more important than ever before, as we can no longer rely on hardware to compensate for inefficient software.

In this work, we will primarily focus on perhaps the fastest-growing of the "big data" domains: genomics [119]. Advances in sequencing technologies have culminated in an unprecedented data explosion within the field. Figure 1-1 shows sequencing costs as a function of time, as compared to projections based on Moore's Law; costs have dropped significantly faster than what Moore's Law would have predicted, and the amount of generated sequencing data has grown exponentially as a result [85]. A tremendous amount of research is consequently being devoted to developing high-

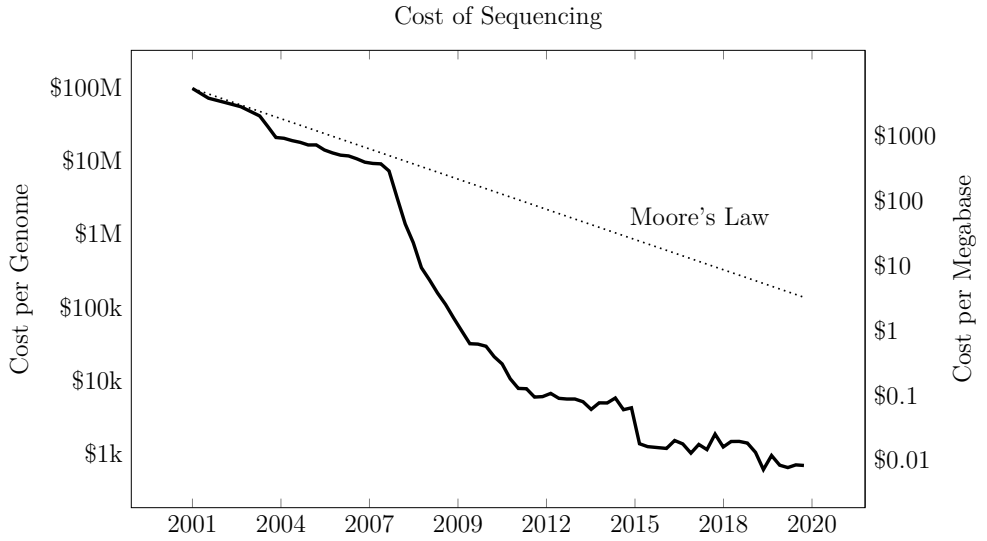


Figure 1-1: Cost of sequencing over time, as compared to projections based on Moore's Law. Results are as of August 2020 (`https://genome.gov/sequencingcosts`).

performance tools and algorithms for genomic data analysis, both from an algorithm design standpoint and a from a software engineering and optimization standpoint [76, 75, 129, 87, 128, 112]. This research ultimately leads to new revelations in biology, medicine, and healthcare.

However, despite relying heavily on optimized, high-performance software, genomics and bioinformatics face a host of critical challenges in terms of software development, testing, and maintenance, which have even resulted in reproducibility issues throughout the field [98, 17, 69]. There are several key reasons for this, including:

- New sequencing technologies and data types are produced regularly, necessitating frequent updates to existing software in order to keep pace, which often leads to corner-cutting and messy, difficult-to-maintain software.

- Datasets continue to grow in size and scope, meaning a tool or algorithm that was developed with a particular scale of data in mind (say, gigabytes) will often need to be fundamentally reworked to be applied to data of a larger scale (say, terabytes).

- Genomics research is inherently exploratory in nature, meaning the algorithms

and tools that researchers use are constantly shifting to experiment with new theories and hypotheses.

- Many researchers in bioinformatics are in fact biologists by training rather than computer scientists or software engineers, and simply lack the expertise to employ software development best-practices.

- The vast majority of bioinformatics software is developed at research universities by 1–2 individuals and is not supported by a dedicated team of engineers, leading to numerous abandoned software projects once the developer(s) graduate or leave the institution.

- Research priorities often clash with software development priorities, as submission deadlines frequently result in rushed software that is not properly tested, and often times does not even work as intended on datasets other than what its authors experimented with.

- There is simply a lack of good software development tools and languages for bioinformatics. High-level languages like Python make development easier but have extremely poor performance on real-world datasets, whereas low-level languages like C or C++ are significantly more difficult to program with and maintain.

1.1 The Need for a New Language

Recent advances in next-generation sequencing (NGS) technologies are continuing the sequencing revolution, and providing a means to study various biological processes through a genomic lens. Because of its novel capabilities and vast scale, NGS has even bigger computational needs, as terabytes of data need to be processed and analyzed with the aid of various novel computational methods and tools [85]. These tools [76, 75, 129, 87, 128, 112] are used on a daily basis in research laboratories and have fueled major discoveries such as establishing mutation-disease links [84] and

detecting recent segmental duplications in the genome [15].

Despite these advances, however, many contemporary genomic pipelines cannot scale with the ever-increasing deluge of sequencing data, which has necessitated impractical and expensive *ad hoc* solutions such as frequent hardware upgrades and constant (re-)implementation of underlying software. Many promising methods are also too difficult to use and replicate because they are often manually tuned for a single dataset, further fueling the recent replication crisis [98, 17]. Finally, many tools are not maintained due to a lack of personnel, expertise, and funds, which has led to many abandoned code repositories that cannot easily be modified to suit researchers' needs, although being vastly superior to the well-maintained alternatives in theory.

The root cause of these problems lies in the general-purpose languages that are used for bioinformatics software development. The most popular programming languages, like Python and R, are not designed to efficiently handle and optimize for sequencing data workflows. Even so, researchers often use such high-level languages to analyze NGS data as they can quickly and easily express high-level ideas, despite a steep performance penalty. Alternatively, a researcher may manually implement low-level optimizations in a language like C. However, doing so requires not only a considerable time investment, but also performance engineering expertise, and often results in hard-to-maintain codebases ridden with subtle bugs and tied to a particular architecture, especially in a field like computational biology where many researchers are not software engineers by trade. These issues are further exacerbated as the field shifts towards the use of third-generation portable sequencers that are powered by resource-limited devices [79], which warrant entirely different software designs and optimizations.

In this thesis, we propose a high-performance, domain-specific programming language (DSL) for bioinformatics and genomics, called **Seq**[1]. Over the course of this work, we will describe why and how a DSL can significantly enhance *both* software performance

[1] `https://seq-lang.org`

and ease of software development, maintenance, and reuse through novel compiler optimizations and language features.

Seq circumvents the tedium of learning a new language by borrowing the syntax, semantics and libraries of the ubiquitous Python[2] programming language, which is used extensively in bioinformatics and beyond [105]—in fact, Python is *the* most popular programming language today by many metrics [100]. While Seq emulates Python, it is a standalone system built entirely from the ground up so as to be statically analyzable, and is thereby able to achieve radically better performance, even outperforming C and C++ in many cases. The ability to efficiently execute Python programs, as well as many of the principles used by Seq, are in fact not unique to bioinformatics, and lend themselves well to *any* domain that requires high-performance. We will also show how Seq can be generalized to a number of different domains, allowing practitioners to write high-level Python code all the while matching or surpassing C/C++'s performance.

The dichotomy between high-performance and ease of use is a longstanding issue in programming, where often one is sacrificed in favor of the other. In this work, we show that this need not be the case—that it is in fact possible to offer high-performance through an interface that is familiar, intuitive, high-level and flexible—thereby democratizing high-performance computing in domains that critically need it.

1.2 Thesis Contributions

This thesis makes the following contributions:

- We introduce novel programming language features and abstractions tailored for bioinformatics, which greatly simplify the representations of common algorithms and patterns in genomics software.

[2] `https://www.python.org`

- We introduce many new genomics-specific compiler optimizations—such as for optimizing fundamental operations like queries of large genomic index or sequence alignment—which are based on the computational domain, its data types and operations, and its algebraic structure.

- We show how high-level, dynamic languages—the foundation of much bioinformatics software—can be statically type checked and compiled to machine code that does not incur runtime overhead nor runtime type information, by sacrificing a small set of language features. This is particularly significant in a field like genomics, where applications often process billions of small sequences concurrently, thus making any per-object overhead problematic.

- We introduce the notion of "bidirectional" compilation and compiler intermediate representations, which are more conducive to writing higher-level optimizations, transformations, and analyses than their alternatives.

- By combining the above contributions, we design and implement Seq, a high-performance, Pythonic, domain-specific language for bioinformatics and genomics.

- We show how Seq can greatly enhance the performance and code simplicity of many common, real-world applications and tools, as well as those of various benchmarks.

- We show how the principles used by Seq—from static type checking to bidirectional compilation—can be generalized to new domains with similar results. To that end, we implement Codon, a domain-configurable compiler framework built on Seq's foundation.

1.3 Thesis Roadmap

The remainder of the thesis is organized as follows:

- Chapter 2 gives background information on computational genomics and on compilers, laying the groundwork for much of what follows.
- Chapter 3 introduces the Seq language, while providing multiple examples of Seq's novel, domain-specific language constructs and features.
- Chapter 4 discusses Seq's type system, and the approaches taken in order to compile Python-like code to machine code without runtime overhead.
- Chapter 5 describes Seq's intermediate representation, optimizations framework, and the ensuing code generation.
- Chapter 6 lays out the many domain-specific optimizations performed by the Seq compiler, as well how several of Seq's domain-specific data types are implemented.
- Chapter 7 describes additional compiler optimizations, including Python-specific and general-purpose ones.
- Chapter 8 shows how Seq can be generalized to new domains by designing a domain-extensible compiler built on Seq's foundation.
- Chapter 9 provides performance results on real-world, end-to-end applications as well as numerous benchmarks, both domain-specific and general-purpose.
- Chapter 10 discusses related work pertaining to multiple facets of this thesis.
- Chapter 11 provides closing remarks and concludes the thesis.

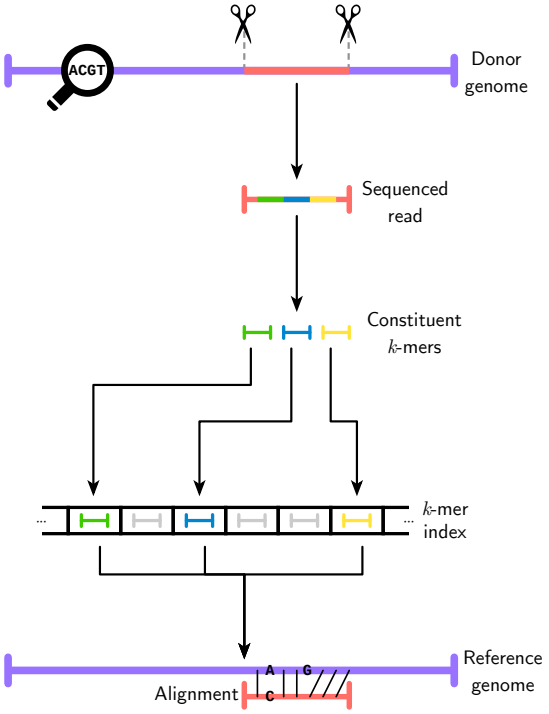
Chapter 2

Background

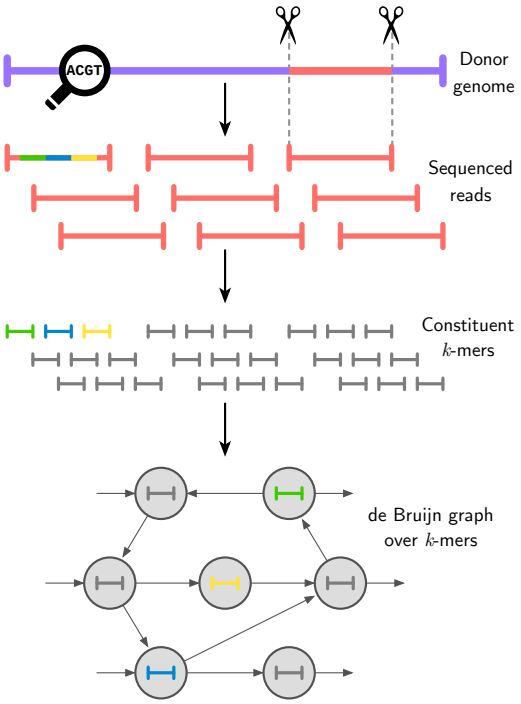
Much of the work presented in this thesis lies at the intersection of computational genomics and compilers. In this chapter, we present the relevant background information on each of these fields.

2.1 A Primer on Computational Genomics

The fundamental data type in computational genomics is the *sequence*, which is conceptually a string over $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$, representing the four nucleotides (also called *bases*) that comprise DNA. Sequences come in several different forms, with varying properties such as length, error profile and metadata. For example, *genome sequencing*—a process that determines the DNA content of a given biological sample—typically produces *reads*: DNA sequences roughly 100 bases in length, with a substitution error rate less than 1% and metadata consisting of a unique identifier and a string of *quality scores*, indicating the sequencing machine's confidence in each reported base of the read. Reads are often analyzed in the context of a *reference genome*, a much longer (in the case of human, 3 gigabase-length) sequence that represents the consensus sequence of an organism's genome in its entirety. A standard first step in nearly any sequence analysis pipeline is *sequence alignment*, which is the process of



(a) Overview of the alignment process for sequencing data. A sequencing machine produces a *read*: a roughly 100 base pair DNA sequence randomly sampled from the donor's genome. Most alignment algorithms then split this read into k-mers—fixed length-k subsequences—and query these k-mers in an index of k-mers from the reference genome to determine candidate alignment positions. Finally, full dynamic programming alignment (typically via an adapted Smith-Waterman algorithm) is carried out to produce the final alignment.



(b) Overview of *de novo* genome assembly from sequencing data. Sequenced reads are partitioned into constituent k-mers, which are then taken to be nodes in a de Bruijn graph whose edges represent $(k-1)$-length overlaps. Other formulations use $(k-1)$-mers (two for each original k-mer) as nodes with the original k-mers represented by the edges. The assembled sequence corresponds to an Eulerian path on this graph.

Figure 2-1: Visualizations of two standard computational genomics applications.

identifying the position in the reference sequence to which a particular read aligns with the smallest edit distance (although many different formulations of this problem exist, such as finding *all* alignments under a given edit distance threshold). To this end, reads are typically first split into fixed length-k contiguous subsequences called *k-mers*, which are then queried in an index of k-mers from the reference to guide the alignment process, as shown in Figure 2-1a. The index itself is an abstract data type that maps k-mers to positions (also called *loci*) in the reference at which they appear, and is often implemented in practice as a hash table or FM-index [77, 49].

Due to the large memory footprints of these structures (roughly 5 gigabytes for optimized FM-indices and tens of gigabytes for hash tables) given the size of the genome, coupled with their poor cache performance, many alignment algorithms spend a significant fraction of their time time stalled on memory accesses; the fraction of stalled cycles in these applications can be over 70% depending on the input dataset [10]. Once a candidate locus is found via the index (and possibly after several filtering steps), a full dynamic programming alignment is performed, usually via a variant of the Smith-Waterman algorithm. Because dynamic programming alignment is a key kernel in nearly all alignment algorithms, there has been substantial research into designing hand-optimized implementations that exploit SIMD vectorization for better performance [48, 124, 120]. One additional complication in sequence alignment is that, while half of all reads will align in the so-called *forward direction* (i.e. without modification), the other half will only align in the *reverse* direction, meaning the read must be *reverse complemented* before alignment. Reverse complementation of a sequence is an operation where the sequence is reversed, and A-bases are swapped with T-bases while C-bases are swapped with G-bases (and vice versa). The fact that half the reads are reverse complemented with respect to the reference genome is a biproduct of the double-stranded nature of DNA, and ultimately leads to reverse complementation being a very common operation that is done on sequences.

Alongside alignment, another common application in computational genomics is *de novo* assembly, where the reads are used to "reconstruct" the donor genome, in the

absence of a predefined reference sequence. While several approaches to this problem exist, perhaps the most common is to again partition the reads into k-mers, build a de Bruijn graph whose vertices are these k-mers with edges indicating that a given k-mer overlaps with another, and finally to find an Eulerian path through this graph, which would encode the assembled sequence [60]. An overview of this process can be seen in Figure 2-1b. As in alignment, there are several additional steps involved in practice, such as counting and filtering k-mers, as well as error correction (as assembly is more sensitive to errors than alignment) [113].

Looking further downstream in the genomic analysis pipeline, computational biologists employ a slew of techniques to handle the problems at hand. However, virtually any downstream model or algorithm, regardless of its domain (machine learning, graph algorithms, etc.), is built on top of the sequence manipulation building blocks described above. For example, structural variation detection (the discovery of novel genomic rearrangements) starts by analyzing read alignment irregularities to detect potential breakpoints of a rearrangement, and proceeds by correcting those alignments via more advanced read alignment schemes that utilize k-mers and FM-indices. Many other problems, such as mutation calling, gene copy number variation detection, genome-wide association studies and cancer driver identification, proceed in a similar fashion. Thus, the common threads between alignment, assembly and many other applications in the genomics domain are the data types used (i.e. various types of sequences like reads, reference or fixed-length k-mers) and the low-level operations performed on them (i.e. some form of matching, indexing, splitting sequences into subsequences or k-mers, reverse complementation, etc.). However, these operations are often embedded in vastly different higher-level algorithms; compare, for example, the dynamic programming involved in alignment and the de Bruijn graph path finding involved in assembly.

2.2 A Primer on Compilers

Compilers in general are exceedingly complex pieces of software that undertake a number of different tasks and operations [39]. At the most abstract level, compilers are programs that convert code from a *source* language to a *target* language. For example, a C compiler would typically convert C source code into assembly language or machine code. Sometimes, compilers are a part of larger systems that are responsible for executing the input program as well; for example, Python uses a compiler to convert Python programs into *bytecode*, a simplified, linear representation of the program, which is then executed by the Python virtual machine. Importantly, compilers must preserve the semantic meaning of the program they are translating: the assembly code generated by a C compiler must adhere to the original C program's behavior as dictated by the C language specification. Further, compilers typically perform code optimizations or analyses to improve on the source program in some way, without altering its semantics. The objective of these could be to reduce code size, make the program faster, limit memory footprint, or anything else.

With these goals in mind, most compilers are designed with the following high-level components.

2.2.1 Front-end

A compiler front-end is responsible for *parsing* the input source program. Thereby, it is typically converted to an *abstract syntax tree* (AST), a tree representation of the original code that retains the program's semantics but strips away superfluous language features like the notion of operator precedence, parenthesis, indentation, and more. Many front-ends also perform *type checking*, which is the process of assigning types to AST nodes. For example, the AST node for the `+` operator in the expression `2 + 2` would be assigned the type `int` by the C compiler's type checker, whereas `2 + 2.5` would be assigned type `double`, representing a floating-point type.

2.2.2 Mid-end

While they are a simplification of the original program, ASTs nevertheless still often include many more types of nodes than what is necessary to capture the source code's semantics. For example, many different types of nodes in an AST might be logically equivalent to a single operation or construct. C arrays, for instance, are in fact logically equivalent to pointers, potentially leading to multiple AST representations of the same concept. For this reason, many compilers contain a mid-end that converts the AST to a simplified representation—called an *intermediate representation* (IR)—that consists of a much more limited set of nodes or constructs.

IRs are particularly useful for program analyses, transformations, and optimizations, and the compiler mid-end is usually where these operations are performed. These operations are collectively referred to as *passes*. Passes might depend on other passes; for example, a pass to eliminate dead code might rely on a pass to fold constants so as to determine regions of code that will never be executed. Many IRs thus include infrastructure for writing, scheduling, and running passes.

Integral to IRs and IR passes is the concept of *lowering*, which is the process of eliminating certain node types by replacing them with other semantically equivalent IR structures. For example, a `for`-loop in C might have a dedicated IR node, but could be lowered to a series of branches or `goto`s during compilation. Lowering is important because certain passes only make sense at a particular level of abstraction, i.e. before or after certain nodes have been lowered. After the loop lowering in the previous example, for instance, it is much more difficult to reason about the loop parameters like bounds or step size, and thus optimizations like loop tiling or strip-mining become impractical—these optimizations would therefore be performed prior to loop lowering. On the other hand, various control flow analyses might be more practical if higher-level control-flow structures have already been lowered to branches or `goto`s, meaning they might be performed *after* loop lowering.

2.2.3 Back-end

Finally, the compiler back-end is responsible for actually generating code for the target, be it machine code, assembly, or something else. This process is highly target-specific. Generating assembly code, for example, requires instruction selection, register allocation, and instruction scheduling, all of which are highly dependent on the target's features and instruction set.

As a result, a number of software libraries have been developed to facilitate code generation for a number of targets. These libraries include LLVM [67], libJIT[1], and GNU lightning[2].

2.2.4 Domain-specific compilers

While general-purpose compilers are able to optimize and generate code for programs across a wide range of applications, there are often missed optimization opportunities that stem from the specific application area or domain. Since general-purpose compilers have no knowledge of domain-specific data types or operations, they are unable to leverage properties of the domain to further optimize code. For instance, consider an application that works with large matrices, and a series of operations therein that results in a double-transpose of a large matrix—a general-purpose compiler has no way of knowing that transposing a matrix twice has no effect, whereas a domain-specific compiler can use the algebraic properties of the transpose operator to do away with the operation entirely.

Hence, domain-specific compilers can often attain better performance than general-purpose compilers in their respective domains, by incorporating *domain-specific optimizations*. Such compilers are often accompanied by new or extended *domain-specific languages* that are specifically tailored for the domain of interest, with the goal of simplifying the programming and debugging process within the domain. More specifically, a domain-specific language enables the compiler to recognize and optimize the

[1] `https://www.gnu.org/software/libjit`
[2] `https://www.gnu.org/software/lightning`

idioms and patterns that are common within the domain, and at the same time makes the programmer's job easier by allowing them to directly express these concepts at a high level, without worrying about low-level details.

The primary focus of this thesis is on designing a domain-specific language and compiler for bioinformatics and computational genomics. Although genomics has seldom been looked at through the lens of compiler theory and design, we show in this work that there is much to be gained by doing so.

Chapter 3

The Seq Language

In this chapter, we introduce Seq, a domain-specific language (DSL) and compiler designed to provide productivity *and* high-performance for computational biology. Seq is a subset of Python, and therefore provides Python-level productivity; yet, the compiler can generate efficient code because the language is statically-typed with compile time support for Python's duck typing. Seq provides data types tailored to computational genomics and uses domain-specific information to optimize code. The Seq DSL allows computational biology experts to quickly prototype and experiment with new algorithms as they would in Python, without imposing the burden of learning a new language. Further, Seq is designed to hide all low-level, complex code optimizations from the end user. Unlike libraries, the Seq compiler can perform optimizations such as operator fusion and pipeline transformations, which we demonstrate to have a substantial benefit.

In particular, this chapter does the following:

- We introduce Seq, a domain-specific language and compiler for computational biology.
- We introduce Seq's genomics-specific data types (e.g. sequence and k-mer types) and operators (e.g. for reverse complementation, k-merization, etc.), fur-

```
from sys import argv
from bio import *
from genomeindex import *

# k-mer for indexing
K = Kmer[20]

# create index over 20-mers
index = GenomeIndex[K](argv[1])

# lookup k-mer in index
@prefetch
def process(kmer, index):
    # forward lookup:
    hits_fwd = index[kmer]
    # reverse lookup:
    hits_rev = index[~kmer]
    ...  # filter, I/O, etc.

# sequence-processing pipeline
(FASTQ(argv[2])
  |> kmers[K](step=10)
  |> process(index))
```

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
#include "GenomeIndex.h"

char revcomp(char base) {
  switch (base) {
    case 'A': return 'T';
    case 'C': return 'G';
    case 'G': return 'C';
    case 'T': return 'A';
    default: return base;
  }
}

void revcomp(char *kmer, int k) {
  for (int i = 0; i < k/2 + k%2; i++) {
    char a = revcomp(kmer[i]);
    char b = revcomp(kmer[k - i - 1]);
    kmer[i] = b;
    kmer[k - i - 1] = a;
  }
}

void process(char *kmer, int k,
             GenomeIndex &index) {
  auto hits_fwd = index[kmer];
  revcomp(kmer, k);
  auto hits_rev = index[kmer];
  revcomp(kmer, k);  // undo
  ...
}

int main(int argc, char *argv[]) {
  const int k = 20;
  const int stride = 10;
  auto *index = GenomeIndex(argv[1], k);
  std::ifstream fin(argv[2]);
  std::string read;
  long line = -1;
  while (std::getline(fin, read)) {
    line++;
    // skip over non-sequences in FASTQ
    if (line % 4 != 1) continue;
    auto *buf = (char *)read.c_str();
    int len = read.size();
    for (int i = 0; i + k <= len; i += stride)
      process(kmer, k, index);
  }
}
```

Figure 3-1: Example k-merization and seeding application in Seq and C++.

ther augmented with additional language constructs such as compiler-optimized pipelines and genomic matching, to both simplify the algorithmic descriptions of complex problems and to enable domain-specific optimizations.

- We show how, in just a few lines of high-level Pythonic code, Seq can express many complex operations and algorithms.

3.1 Seq at a Glance

Seq provides built-in language-level facilities for seamlessly expressing many of the types and design patterns found in genomics applications. As an example, consider reading a set of sequencing reads from a FASTQ file (a standard format for storing

```
from sys import argv
from bio import *

K = Kmer[16]
counts = {}

for record in FASTQ(argv[1]):
  read = record.read
  for kmer in read.kmers[K](step=1):
    counts[kmer] = counts.get(kmer, 0) + 1

ordered = sorted((v,k) for k,v in counts.items())
for kmer,count in ordered:
  print(count, kmer)
```

Figure 3-2: Example k-mer counting application in Seq. The shown code prints unique k-mers from a FASTQ file in ascending order by count.

reads) and querying each read's constituent k-mers in a genomic index. This process is commonly referred to as *seeding*, and is the first step in nearly any sequence alignment algorithm [77].

An implementation of 20-mer seeding in Seq is shown in Figure 3-1. Seq uses the familiar syntax of Python, but incorporates several genomics-specific features and optimizations. k-mer types like `Kmer[20]` (which represents a k-mer with 20 bases), for example, allow for easy k-merization (the process of splitting a sequence into k-mers, done using `kmers` in Seq) and reverse complementation (using `~kmer`). Similarly, pipelining—a natural model for thinking about processing reads—is easily expressible in Seq, where a user can define pipelines via the `|>` operator as shown in the figure. Seq can also speed up expensive index queries via pipeline transformations that allow for effective software prefetching (`@prefetch` annotation). Compare this Seq implementation to the C++ implementation also shown in Figure 3-1, which includes extensive boilerplate code for reverse complementation and FASTQ iteration, and cannot perform the domain-specific pipeline or encoding optimizations made by the Seq compiler, which in practice we find to attain upwards of 1.5–2× speedups over optimized C++ implementations (Chapter 9).

Construct	Meaning
`seq`	Sequence type
`Kmer[`k`]`	k-mer type
`~s`	Reverse complement of `s`
`s[i:j]`	Subsequence of `s` from index `i` to `j`
`Kmer[`k`](s)`	Conversion of `s` to a k-mer
`s.split(k, step)`	Iterator over length-`k` subsequences of `s` with given step size
`s.kmers[K](step)`	Iterator over k-mers of type `K` in `s` with given step size
`a |> b |> c`	Pipeline with stages `a`, `b`, and `c`
`a ||> b |> c`	Same as above, except output of `a` processed in parallel
`case 'A*G'`	Pattern matching sequences starting with `A`, ending in `G`
`import bio`	Usage of Seq's bioinformatics library

Table 3.1: Summary of some of Seq's genomics-specific language constructs.

As a second example, consider the code in Figure 3-2, which shows a simple k-mer counting application; k-mer counting is an extremely common operation in genomics and bioinformatics, the goal of which is to obtain the frequency of each k-mer in a dataset [83]. Notice that, in Seq, domain-specific types and functions—e.g. sequences, k-mers, iterators over genomic files—can interact seamlessly with standard Python types and functions—e.g. dictionaries, sorting, iterators—as in this example.

A summary of some of Seq's domain-specific language constructs is given in Table 3.1. In the upcoming sections, we will discuss each of these constructs in more detail, and show how they can be used to easily express a range of common idioms, patterns, and operations within genomics applications.

3.2 Design Goals

A critical barrier to any new language's success in a particular field is its initial adoption, as most potential users already have a set of languages, environments and packages with which they are comfortable. This is particularly true in bioinformatics, where many researchers are biologists first and programmers second. For this reason, the Seq language borrows the syntax and semantics of Python—one of the most widely-used languages in bioinformatics—and adds numerous genomics-oriented

language features and constructs. Indeed, most of the preexisting Python code that is used within the genomics community will compile and run without modification in Seq, ultimately allowing the user to attain the performance of C/C++ with the programming ease of Python.

To achieve this, we designed a compiler with a static type system. It performs Python-style duck typing and runtime type checking at compile time, completely eliminating the substantial runtime overhead imposed by the reference Python implementation, CPython[1], and most other Python implementations alike. Unlike these, we reimplemented all of Python's language features and built-in facilities from the ground up, completely independent of the CPython runtime. The Seq compiler uses an LLVM [67] backend, and in general uses LLVM as a framework for performing general-purpose optimizations. Seq programs additionally use a lightweight runtime library for I/O and memory allocation. We describe Seq's domain-agnostic implementation details in depth in Chapters 4 and 5.

3.3 Sequences and k-mers

Seq's namesake type is indeed the sequence type: `seq`. A `seq` object represents a DNA sequence of any length and—on top of general-purpose string functionality—provides methods for performing common sequence operations such as splitting into subsequences, reverse complementation and k-mer extraction. Alongside the `seq` type are k-mer types, which are dependent on the k-mer length. For example, `Kmer[1]` represents a 1-mer, `Kmer[2]` a 2-mer and so on, up to `Kmer[1024]` (a reasonable upper bound on k-mer length in nearly any genomics application).

Sequences can be seamlessly converted between these various types, as shown in Figure 3-3. In fact, this pattern is prevalent in many genomics applications, where longer sequences (be it a read, reference or anything else) are split into their constituent k-mers, and each is subsequently processed.

[1] "Python" is henceforth used as a synonym for "CPython" unless otherwise specified.

```
from bio import *
dna = s'ACGTACGTACGT'  # sequence literal

# (a) split into subsequences of length 3
#     with a stride of 2
for sub in dna.split(k=3, step=2):
    print sub

# (b) split into 5-mers with stride 1
FiveMer = Kmer[5]
for kmer in dna.kmers[FiveMer](step=1):
    print kmer
    print ~kmer  # reverse complement

# (c) convert entire sequence to 12-mer
kmer = Kmer[12](dna)
```

Figure 3-3: Example of `seq` and k-mer type usage.

3.4 Pipelines and Partial Calls

Pipelining is a natural model for thinking about processing genomic data, as sequences are typically processed in stages (e.g. read from input file → split into k-mers → query k-mers in index → perform full dynamic programming alignment → output results to file), and are almost always independent of one another as far as this processing is concerned. Because of this, Seq supports a pipe operator: `|>`, similar to F#'s pipe and R's `magrittr` (`%>%`) [12]. Pipeline stages in Seq can be regular functions or generators. In the case of standard functions, the function is simply applied to the input data and the result is carried to the remainder of the pipeline, akin to F#'s functional piping. If, on the other hand, a stage is a generator, the values yielded by the generator are passed lazily to the remainder of the pipeline, which in many ways mirrors how piping is implemented in Bash. Note that Seq ensures that generator pipelines do not collect any data unless explicitly requested, thus allowing the processing of terabytes of data in a streaming fashion with no memory and minimal CPU overhead.

An example of pipeline usage is shown in Figure 3-4, which shows the same two loops from Figure 3-3, but as pipelines. First, note that `split` is a Seq standard library

```
from bio import *
dna = s'ACGTACGTACGT'  # sequence literal

# (a) split into subsequences of length 3
#     with a stride of 2
dna |> split(..., k=3, step=2) |> echo

# (b) split into 5-mers with stride 1
def f(kmer):
    print kmer
    print ~kmer

FiveMer = Kmer[5]
dna |> kmers[FiveMer](step=1) |> f
```

Figure 3-4: Example of pipeline usage in Seq, where the two loops from Figure 3-3 are represented as pipelines.

function that takes three arguments: the sequence to split, the subsequence length and the stride; `split(..., 3, 2)` is a partial call of `split` that produces a new single-argument function f where $f(x) =$ `split(`x`, 3, 2)`. The undefined argument(s) in a partial call can be implicit, as in the second example: `kmers` (also a standard library function) is a generic function parameterized by the target k-mer type and takes as arguments the sequence to k-merize and the stride; since just one of the two arguments is provided, the first is implicitly replaced by `...` to produce a partial call (i.e. the expression is equivalent to `kmers[FiveMer](..., 1)`). Both `split` and `kmers` are themselves generators that yield subsequences and k-mers respectively, which are passed sequentially to the last stage of the enclosing pipeline in the two examples.

3.4.1 Parallelism

CPython and many other implementations alike cannot take advantage of parallelism due to the infamous global interpreter lock, a mutex that protects accesses to Python objects, preventing multiple threads from executing Python bytecode at once [20]. Unlike CPython, Seq has no such restriction and supports full multithreading. To

```
from bio import *
dna = s'ACGTACGTACGT'  # sequence literal

# (a) split into subsequences of length 3
#     with a stride of 2
dna |> split(..., k=3, step=2) ||> echo

# (b) split into 5-mers with stride 1
def f(kmer):
    print kmer
    print ~kmer

FiveMer = Kmer[5]
dna |> kmers[FiveMer](step=1) ||> f
```

Figure 3-5: Example of parallel pipeline usage in Seq, where the two pipelines from Figure 3-4 are parallelized.

this end, Seq supports a *parallel* pipe operator `||>`, which is semantically similar to the standard pipe operator except that it allows the elements sent through it to be processed in parallel by the remainder of the pipeline. Hence, turning a serial program into a parallel one often requires the addition of just a single character in Seq, as shown by Figure 3-5. Further, a single pipeline can contain multiple parallel pipes, resulting in nested parallelism.

Internally, the Seq compiler uses Tapir [108] with an OpenMP task backend to generate code for parallel pipelines. Logically, parallel pipe operators are similar to parallel for-loops in OpenMP: the portion of the pipeline after the parallel pipe is extracted into a new function that is called by the OpenMP runtime task spawning routines (as in `#pragma omp task` in C++), and a synchronization point (`#pragma omp taskwait`) is added after the outlined segment. Lastly, the entire program is implicitly placed in an OpenMP parallel region (`#pragma omp parallel`) that is guarded by a "single" directive (`#pragma omp single`) so that the serial portions are still executed by one thread (this is required by OpenMP as tasks must be bound to an enclosing parallel region).

```
def describe(n):
    match n:
        case m if m < 0:
            print 'negative'
        case 0:
            print 'zero'
        case m if 0 < m < 10:
            print 'small'
        case _:
            print 'large'
```

Figure 3-6: Example usage of `match`.

3.5 Pattern Matching

Seq provides the conventional `match` construct, which works on integers, lists, strings and tuples. An example usage of `match` is shown in Figure 3-6. A novel aspect of Seq's `match` statement is that it also works on sequences, and allows for concise recursive representations of several common sequence operations such as subsequence search, reverse complementation tests, and base counting, which are shown in Figure 3-7. Sequence patterns consist of literal `ACGT` characters, single-base wildcards (`_`) or "zero or more" wildcards (`*`) that match zero or more of any base.

3.6 The `bio` Module

Lastly, Seq includes a built-in `bio` library that includes implementations of numerous standard genomics algorithms and data structures, all with intuitive, Pythonic APIs. The `bio` module is implemented entirely in Seq itself, and includes:

- Parsers for many standard file formats, including FASTQ, FASTA, SAM, BAM, CRAM, VCF, BED, and more.
- SIMD-optimized functions for sequence alignment, which are further accelerated through domain-specific compiler optimizations (Chapter 6).
- FM- and FMD-index data structure implementations, along with utilities for

```
# (a)
def has_spaced_acgt(s):
    match s:
        case 'A_C_G_T*':
            return True
        case t if len(t) >= 8:
            return has_spaced_acgt(s[1:])
        case _:
            return False

# (b)
def is_own_revcomp(s):
    match s:
        case 'A*T' | 'T*A' | 'C*G' | 'G*C':
            return is_own_revcomp(s[1:-1])
        case '':
            return True
        case _:
            return False

# (c)
def count_bases(s):
    from bio.seq import BaseCounts
    match s:
        case 'A*': return count_bases(s[1:]).add(A=True)
        case 'C*': return count_bases(s[1:]).add(C=True)
        case 'G*': return count_bases(s[1:]).add(G=True)
        case 'T*': return count_bases(s[1:]).add(T=True)
        case _: return BaseCounts()
```

Figure 3-7: Example usages of `match` on sequences in Seq. Example (a) checks if a given sequence contains the subsequence `A_C_G_T`, where `_` is a wildcard base; such an operation may be present in an application that uses *spaced seeds*—non-contiguous k-mers that are shown to improve accuracy in some settings [64]. Example (b) checks if the given sequence is its own reverse complement, which is useful in certain sequence hashing schemes [94]. Finally, example (c) counts how many times each base appears in the given sequence, which can e.g. be used to determine GC content (the fraction of bases that are G or C) [114].

```
from C import sqrt(float) -> float
from C import puts(ptr[byte])
print sqrt(100.0)
puts("hello world".c_str())
```

Figure 3-8: Example of external function usage in Seq with C standard library functions `sqrt` and `puts`.

computing Burrows-Wheeler Transforms and suffix arrays.

- Utilities for working with genomic loci and intervals.
- General DNA sequence operations and functionality (e.g. subsequence, iteration, reverse complementation, etc.) as well as support for protein sequences and operations on them (e.g. alignment, translation, etc.).

A more comprehensive description of the `bio` module's contents is given in Appendix C.

3.7 Other Features

In addition to its genomics-specific features, Seq includes a number of general-purpose features that aim to make the language more flexible and interoperable.

3.7.1 External functions

Seq enables seamless interoperability with C and C++ via externally-imported functions, as shown in Figure 3-8. Primitive types like `int`, `float`, `bool` etc. are directly interoperable with the corresponding types in C/C++, while compound types like tuples are interoperable with the corresponding struct types. Other built-in types like `str` provide methods to convert to C analogs, such as `c_str()` as shown in Figure 3-8.

3.7.2 Type extensions

Seq provides an `@extend` annotation that allows programmers to add and modify

```
@extend
class int:
    def to(self: int, other: int):
        for i in range(self, other + 1):
            yield i

    def __mul__(self: int, other: int):
        print 'caught int mul!'
        return 42

for i in (5).to(10):
    print i  # 5, 6, ..., 10

# prints 'caught int mul!' then '42'
print 2 * 3
```

Figure 3-9: Example of type extension in Seq.

methods of various types within the current module at compile time, including built-in types like `int` or `str`. This allows much of the functionality of built-in types to be implemented in Seq as type extensions in the standard library. Figure 3-9 shows an example where the `int` type is extended to include a `to` method that generates integers in a specified range, as well as to override the `__mul__` magic method to "intercept" integer multiplications. In short, `@extend` provides a mechanism to extend existing types with additional functionality, such as supporting new operators or conforming to a new API à la Python's duck typing. Note that all type extensions are performed strictly at compile time and incur no runtime overhead.

3.8 Differences with Python

Python is an interpreted language, and does not check for type consistency until necessary during runtime—even then, only the existence of methods required by a given program is checked, an approach commonly referred to as *duck typing*. This simple and clean design, together with a well-thought-out syntax, enables rapid prototyping and a great deal of flexibility without imposing artificial language design constraints, which is partly what has made Python popular in many different domains, bioinfor-

matics notwithstanding.

However, this dynamism comes with a hefty price in terms of performance, as almost any method invocation or variable reference requires expensive dictionary lookups during runtime. Furthermore, the lack of type annotations and the dynamic nature of objects necessitate delaying type checks until a given object is actually used, which can sometimes be *days* after the Python script was initially run in the case of long-running programs. This is a common problem in bioinformatics, where many scripts take a long time to complete due to ever-growing input datasets; a problematic section of code that is preceded by a rarely-entered `if`-statement, for example, can result in many hours of wasted runtime when reached. Moreover, this lazy approach to typing requires the developer to include numerous manual type checks and large test suites to ensure type soundness during execution. Python versions 3.6 and later attempt to mitigate this problem with the optional mypy type checker, which adds support for type annotations with ahead-of-time type checks to the core language (and whose syntax we adopted for consistency). However, mypy must still interoperate with the Python runtime, and as such could leave some types ambiguous (e.g. as `Any`), which does not map easily to LLVM IR—the backbone of Seq's optimization framework. PyPy, on the other hand, uses a restricted subset of Python called "RPython" which can be statically typed, but again does not fit our purposes as it performs type deduction at runtime and allows arbitrary non-RPython code to be mixed in. By contrast, the Seq compiler has a complete view of *all* types at compilation time, which it uses to avoid all runtime overhead.

3.8.1 Basic types and metadata overhead

Python has a relatively simple type system in which all types derive from the `object` base type. Some primitive types (such as integers and floats) are, for performance reasons, implemented directly in C within CPython's runtime. However, even the C implementations of these types carry a significant overhead, as they still have to interoperate with the rest of the Python ecosystem. As can be seen in Figure 3-

10, a simple float object—arguably the most lightweight type Python has—consists of three pointers, an integer and finally the float value itself. This "metadata" is necessary for Python's runtime type resolution and reference counting[2]. For these reasons, all high-performance Python libraries (such as NumPy) achieve their speed by dealing primarily with arrays or matrices that can be abstracted away from the Python runtime to the C level.

Using optimized libraries in this way works well for programs that operate on a small number of very large objects, since the total object metadata is small; however, programs that operate on a very *large* number of *small* objects—even if the total non-metadata memory is the same—suffer from a much larger metadata footprint, which can render them intractable even in the presence of an optimized C library; a visualization is given in Figure 3-11. Genomics and bioinformatics programs typically fall into the latter category, as millions or billions of small sequences are often processed concurrently. Figure 3-12, for example, compares memory usages of a k-mer counting application between C and Python implementations. As the input data size grows, so too does the difference between the two implementations' memory usages; this difference is indicative of the runtime object metadata overhead that Python incurs. Figure 3-13 compares memory usages when creating an increasing number of `double` arrays, while keeping the total non-metadata memory fixed (similar to what is shown in Figure 3-11); once the object count surpasses a given threshold, the metadata overhead begins to grow astronomically. As shown, a domain-specific, optimized library like NumPy can reduce overhead when the number of objects is small, but as the object count increases so does the relative overhead in the NumPy program, due to the large overhead of each individual NumPy array.

To eliminate this metadata overhead, Seq follows a different design philosophy in terms of types. Primitive types such as `int`, `bool`, and `float` map directly to the equivalent LLVM IR types `i64`, `i8`, and `double`, respectively. As such, they incur no

[2]We do note, however, that the `_ob_next` and `_ob_prev` pointers are compiled into the structure definition conditionally, and can be omitted.

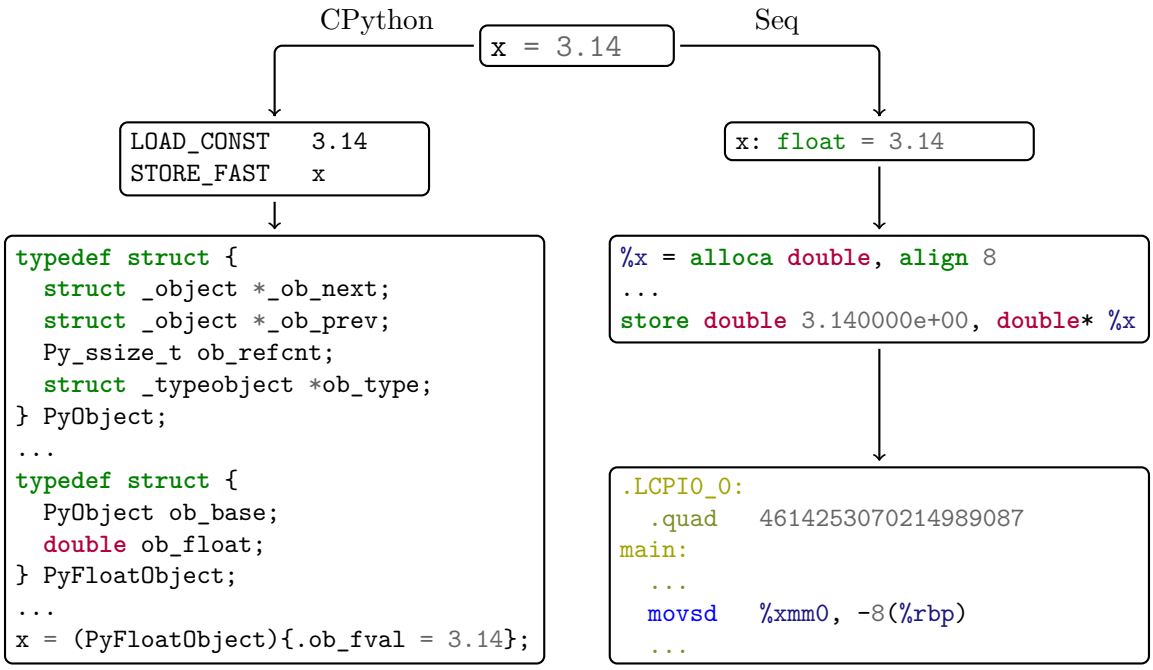


Figure 3-10: Seq versus CPython during compilation and execution of a simple float assignment. CPython compiles to bytecode that omits all type information, and instead relies on runtime type information by virtue of metadata stored alongside the actual `float` value within the `PyFloatObject` structure. By contrast, Seq infers the type of `x` at compile time and compiles the assignment to LLVM IR, which encodes type information. LLVM in turn compiles this to assembly or machine code.

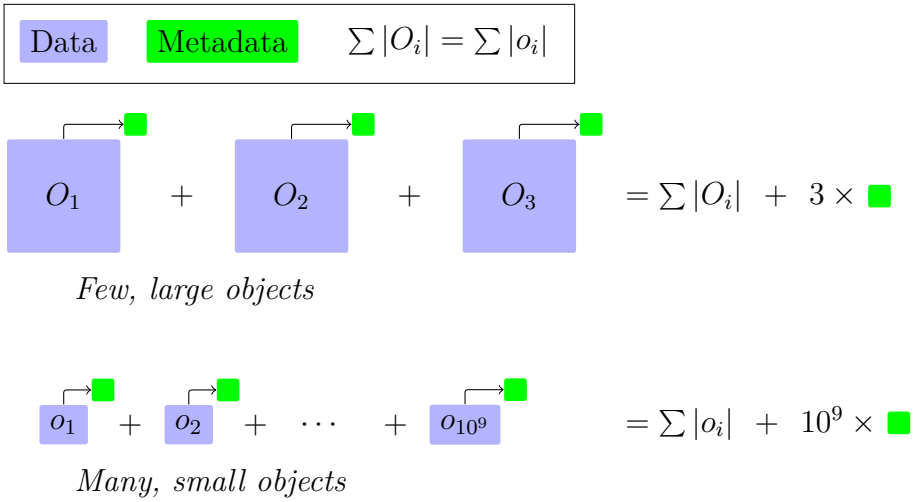


Figure 3-11: Visualization of metadata overhead in two different scenarios: *few, large objects* and *many, small objects*. An example of the former is performing operations on a handful of very large matrices, while an example of the latter is hashing, indexing, and storing billions of k-mers, e.g. for read mapping, assembly, or k-mer counting. The total object data is the same in both cases, but the *many, small objects* case incurs substantially more metadata overhead.

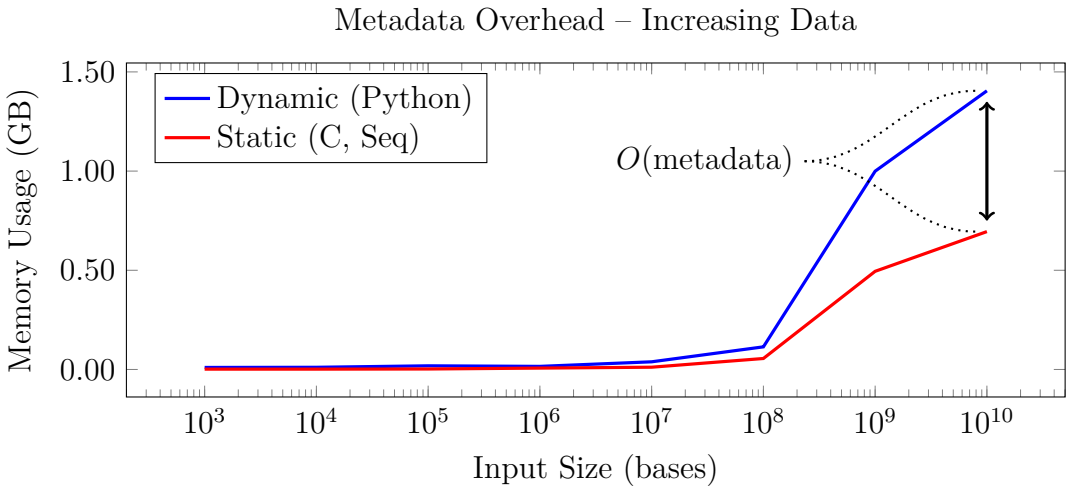


Figure 3-12: Object metadata overhead in Python, using C as a reference. Since Seq has no metadata overhead, its results are identical to C's. The plot shows memory usages for the Python and C implementations of the k-nucleotide benchmark from *The Computer Language Benchmarks Game* suite of benchmarks [53], for various input sizes. k-nucleotide involves counting occurrences of various k-mers in a large FASTA file.

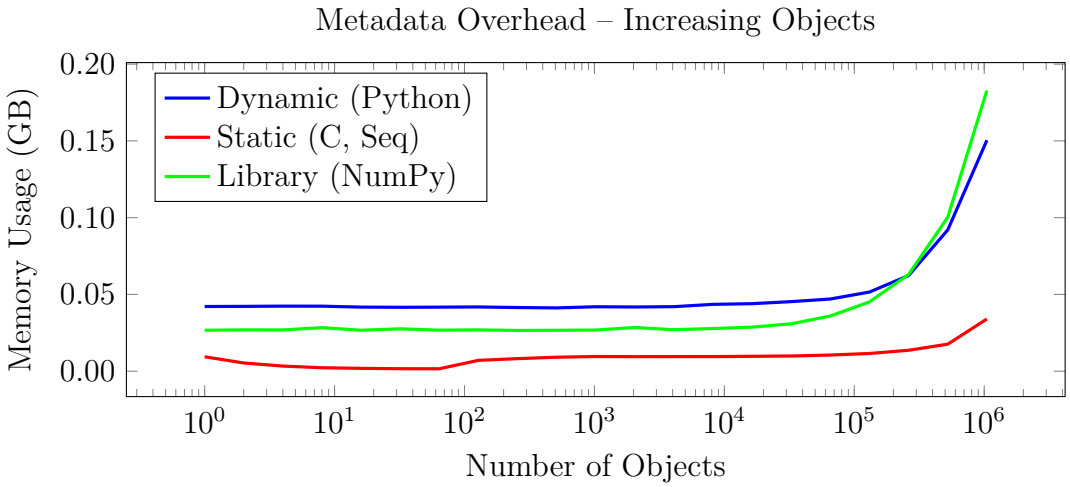


Figure 3-13: Object metadata overhead in Python, using C as a reference. Since Seq has no metadata overhead, its results are identical to C's. The plot shows memory usages from creating an increasing number of `double` arrays (or lists in pure Python) of fixed total size. The difference between C and Python/NumPy is indicative of metadata overhead, especially for larger object counts. NumPy initially has less overhead than pure Python because it stores the array elements inline; when more objects are created, NumPy's overhead grows beyond Python's due to the large overhead each NumPy array carries.

Seq expression	LLVM IR type	Description
`"hello world"`	`{i64, i8*}`	struct of length and character pointer
`(1, 0.5, False)`	`{i64, double, i8}`	struct of tuple element types
`MyClass()`	`i8*`	pointer to heap-allocated `MyClass` struct
`MyClass().foo`	`{i8*, void (i8*)*}`	struct of `self` and method function pointer

Table 3.2: Examples of Seq types mapping to LLVM types. The last example assumes `foo` is defined to be a method of `MyClass` that takes no extra arguments and does not return a value (i.e. `def foo(self: MyClass) -> void`).

overhead whatsoever. Nevertheless, each of these primitives is still logically a fully-fledged type with a set of associated methods that can be extended by the user (e.g. type `int` has a method `__add__` for addition that can be statically patched); there is no overhead as all method dispatches are resolved by the compiler. Furthermore, Seq inlines all magic method invocations on primitive types.

More complex types typically compile to an LLVM aggregate type or a pointer to one. Aggregate types are used in place of Python's tuples and named tuples, and are fully isomorphic to C structs. Pointers to aggregate types—or, more precisely, *reference types*—are used to implement Python classes. As usual, aggregates are passed by value while reference types are passed, unsurprisingly, by reference. Table 3.2 gives a few examples of type conversions from Seq to LLVM.

In order to maintain compatibility with Python, class members can be deduced automatically by lexically analyzing a given class's methods. Python's built-in collection types—`List`, `Set` and `Dict`—are all modeled as reference types in Seq and bootstrapped as standard library classes implemented in Seq itself.

3.8.2 Generic functions, methods and types

Python's lack of static typing allows any function to take objects of any type as arguments. This design philosophy does not translate well to strongly-typed compiled languages that lack runtime type information, as they typically require each function to explicitly specify input and output types.

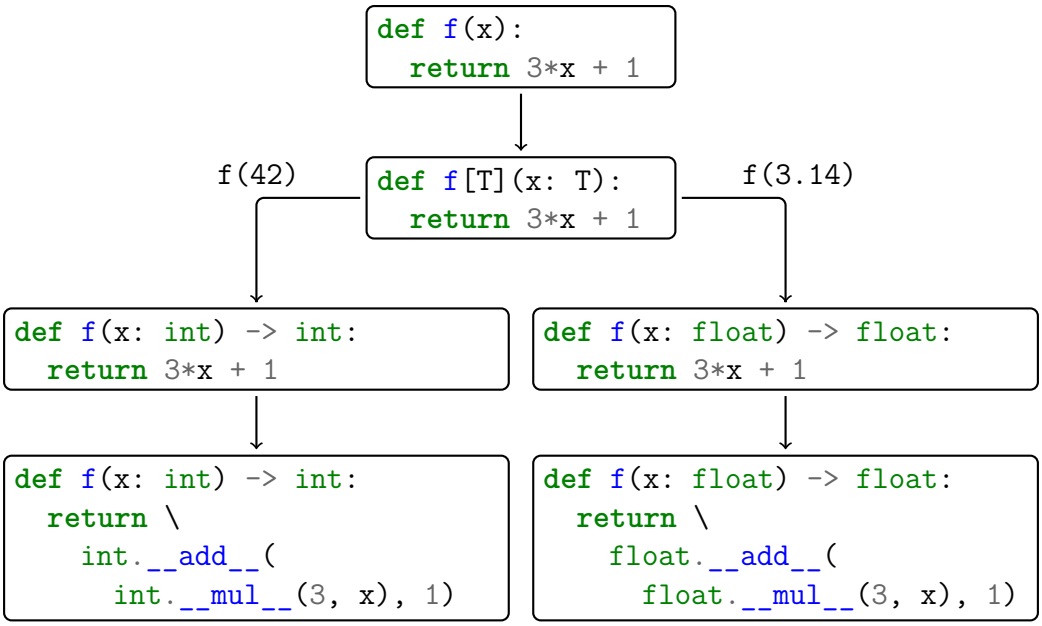


Figure 3-14: Seq's implicit generic type parameters. The function `f` is declared to take a parameter `x` of unspecified type; the Seq compiler treats the type of `x` as generic and clones `f` on demand for each new input type, and subsequently deduces return types.

Code compatibility with Python is of paramount importance for Seq, as it is unreasonable to expect users to manually annotate (or rewrite) their large codebases. Thus, Seq handles this problem by treating each Python function that does not provide type annotations as a *generic function*, where one or more input or output types cannot be deduced from annotations or a lexical analysis of the function body. In this case, each argument without a type annotation (referred to as an *implicit generic*) is replaced by a concrete type on demand at compile time. For example, on encountering `f(42)` as in Figure 3-14, the compiler checks whether there is an instantiation of `f` that accepts an `int` argument, and if so routes the call there. If not, the compiler clones `f`'s AST and creates a new instantiation of the function that specifically accepts an `int` argument (a similar approach is taken by Julia [24]). This newly created function would produce an error if, for example, `int` did not contain an appropriate `__mul__` method as required in the function body. Instantiations are created lazily on demand.

Unlike Python, Seq allows users to explicitly mark functions as generic and to specify explicit generic type parameters, allowing more complex type relationships to be

```
class Node[T]:
    next: Optional[Node[T]]
    data: T

def item[T,U](n: Node[T], f: Function[[T],U]) -> List[U]:
    return [f(n.data)]

n = Node(None, 5)
def foo(x: int) -> str:
    return str(x)
i = item(n, foo)  # type parameters deduced as int and str
i = item[int,str](n, foo)  # explicit specification also OK
```

Figure 3-15: Seq's explicit generic type parameters.

expressed. For example, Figure 3-15 shows a higher-order function that only operates on generic nodes and functions (as `T` is an explicit generic type parameter). The argument types of `item` ensure that the argument function `f` can take the argument node `n`'s data as a parameter. Note that it is impossible for Python-style unnamed generics to cover this use-case without explicit `isinstance` checks. As shown in Figure 3-15, explicit type parameterization is optional even when explicit generics are present, as Seq performs type parameter inference whenever possible.

Analogous reasoning applies to classes, where class members can be generic. Examples of such classes include `List[T]` and `Dict[Key,Value]`. Unlike functions, implicit generics are disallowed in classes as they would impair readability and could lead to ambiguous instantiations during the class member deduction stage. Note that, as far as Seq is concerned, different instantiations of functions and classes are treated as different types. Thus, `f(x: List[int])` and `f(x: List[float])` are represented internally as two separate functions, which allows Seq to optimize each instantiation according to its concrete argument types.

3.8.3 Duck typing

Seq's type system is designed to behave like Python's if one uses Seq as a drop-in Python replacement without specifying explicit types. As long as the methods of every

type are known at compile time (an invariant strictly enforced by Seq as it does not allow type modifications at *runtime*), the compiler will deduce the argument/return types of all methods and instantiate any generic method as appropriate. Indeed, we find that this static instantiation-on-demand simulates duck typing reasonably well. Explicit type annotations enforce an extra layer of typing discipline on top of duck typing (à la mypy), and as such coexist peacefully with it.

3.8.4 Type inference

Any strongly typed language needs a way to infer the type of each variable present in a given program. Languages such as C or Pascal require end users to manually annotate each variable with a type. Other languages, such as C++11 or newer versions of Java, support uni-directional type inference by automatically deducing types of left-hand side terms based on right-hand side types. Initial versions of Seq also used uni-directional type inference, allowing users to write, for instance, `x = 5` instead of `x: int = 5`.

However, uni-directional type inference is unable to handle a few common constructs in the Python language, including empty lists (e.g. `a = []`), nullables (e.g. `a = None`) and lambda functions (e.g. `lambda x: x+1`). With uni-directional inference, each of these constructs requires the user to provide manual type annotations (e.g. `a: List[int] = []`) even if the type can be inferred later. Because of this, Seq uses bi-directional type inference, implemented on top of the Hindley-Milner inference algorithm [57, 88, 44], to automatically annotate such types. We slightly modified the standard Hindley-Milner algorithm to support generic classes, functions and instantiations on demand. We also enforce an invariant where all types within a scope (be it a function scope, class scope or the top-level scope) must be fully deduced by the end of that scope. This implies that a function cannot return a non-instantiated generic type: `def f(): return []`, for example, will cause a compilation error, but `def f[T]() -> List[T]: return []` will compile successfully. Any weakly typed variable or lambda is instantiated as soon as possible (note that Seq treats lambdas

as weakly typed constructs and does not generalize them—generalizations are only applied to generic functions defined with `def` and generic classes). Seq's type system is described in depth in Chapter 4.

3.8.5 Limitations

The strongly-typed nature of Seq does come with some limitations compared to conventional Python. Since all types must be fixed at compile time, a Seq program cannot (for example) create a collection of elements (e.g. `List`) with varying types (this is theoretically possible to support by promoting the list's element type to a union type over the various different element types, although this has yet to be implemented). Seq also does not support method or class monkey-patching at runtime (but it does support this at compile time, as shown in Section 3.7.2), nor indexing into a heterogeneous tuple with a non-constant index (as the type of the resulting expression would be ambiguous). Our type checker and instantiation algorithm also require each function to have a single return type. Finally, while Seq supports class extensions, it does not support subtyping (nor, therefore, fully-fledged polymorphism), meaning that `class A; class B(A)` will copy `A`'s methods to `B` without making `B` a subtype of `A` per se. With these trade-offs, Seq can perform all type-checking at compile time without sacrificing any runtime cycles for type enforcement, and without significantly hindering the expressibility of Python's syntax. We have found that, especially in bioinformatics software, these language capabilities are seldom required (or at least can almost always be replaced by Seq-conforming alternatives with minimal effort); indeed, we are not aware of any genomics application that directly relies on such features. Consequently, these features are omitted in Seq at the time of writing. A summary of Seq's current limitations and differences with Python can be found in Appendix C.2.

It is important to note that some of these limitations can be overcome with additional engineering effort, whereas others might require foundational changes. For example, mixing objects of different types—be it in a collection or by assignment to

a single variable—can be handled by implicitly creating union types; for example, the element type of the list `[42, 'abc']` could become `Union[int,str]`. On the other hand, features such as dynamic polymorphism or subtyping would likely require more substantial changes to the type checking algorithm, and are known to be difficult problems in the context of a Hindley-Milner type system [115].

3.9 Conclusion

We have introduced Seq's language features—both domain-specific and general-purpose—and motivated Seq's adoption of Python's syntax and semantics. In the upcoming chapters, we will delve deeper into the internals of Seq in terms of type checking, its intermediate representation, compiler optimizations, and more.

Chapter 4

Type System

In this chapter, we start from the observation that the primary reason for the poor performance of many high-level languages lies in their dynamism, which prevents them from being compiled and optimized ahead of time. This, in turn, indicates that the culprit is a lack of complete type information at compile time; whereas low-level languages like C, C++ or Rust have complete knowledge of all data types in a program prior to its execution, most high-level languages defer this information until runtime, resulting in substantial overhead and slowdowns. Prior attempts at remedying this issue through just-in-time compilation or other related means often fall short, as they must still interface with the original implementation's runtime and allow for unresolvable types [32]. These attempts also strive to support every single feature of the original language, even though many of the "problematic" features that actually entail dynamism are virtually never used in scientific and high-performance computing, and can often be done away with. In other words, *"giving people a dynamically-typed language does not mean that they write dynamically-typed programs"* [11]. Given this premise, we present in this chapter a strongly-typed alternative to Python's runtime that can resolve types and select dynamic behaviour *at compile time* to the fullest extent possible without any runtime overhead whatsoever. The goal of this alternative approach is to cover as much of Python's syntax and semantics as possible while

minimizing the set of missing features stemming from a lack of dynamic runtime. In other words, we trade some (often unneeded) dynamism for substantial gains in performance.

Decreased performance is often a fair price to pay for Python's extensibility and rapid development cycle. However, as mentioned, in contexts such as high-performance computing, scientific computation, or big data analytics, even a simple loop construct can introduce enough overhead to render Python code hundreds of times slower than its compiled counterparts. Some Python implementations, such as PyPy [26] or Numba [8], attempt to address these shortcomings through just-in-time (JIT) compilation. PyPy achieves this through RPython [9], a limited subset of Python that is fully analyzable and allows complete type inference, and as such is an ideal target for JIT compilation. However, despite its success, RPython is a rather narrow subset of Python, and does not support some advanced use cases (usually involving nested generators). Numba, on the other hand, is geared primarily for optimizing numeric computation and is thus severely limited in scope. Other promising approaches, such as Starkiller [106], are either unavailable or are abandoned, and do not support more advanced Python constructs (such as exceptions and generators).

To remedy these shortcomings, we propose a new type system for Seq, *localized type system with delayed instantiation (LTS-DI)*, which builds on top of the classical Hindley-Milner-Damas bidirectional type inference algorithm used in Standard ML and many other functional languages. While the rules of this system share many similarities with ML-like systems, they also significantly depart from the "canonical ML" rules in order to better support type checking Python-like programs. LTS-DI is one of the key factors that allows Seq to attain 10–100× speedups over standard Python (Chapter 9).

This chapter contributes the following:

- We present an ML-like type system, called *localized type system with delayed instantiation* (LTS-DI), in the context of the Python language.

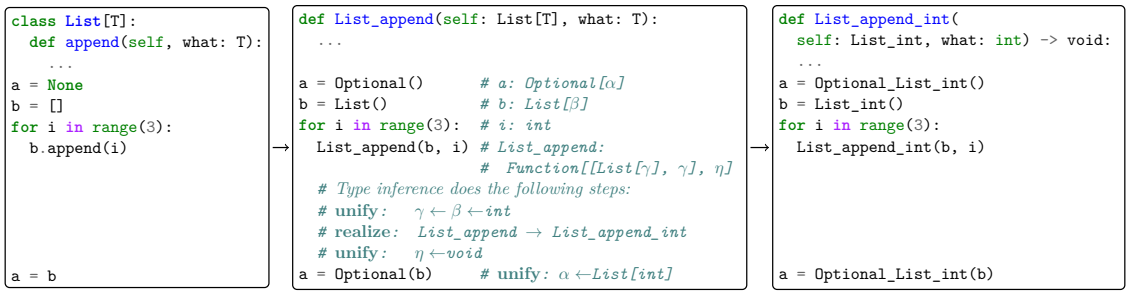


Figure 4-1: Example of type inference and function instantiation performed by LTS-DI. The original Python code on the left gets transformed to the fully type checked version on the right. The middle box shows the first type checking pass and annotates type assignments, unification (type merging) and function realization.

- We show how to extend the Hindley-Milner-Damas's unification algorithm for bidirectional type inference with additional, novel features—including type localization, delayed function instantiation, monomorphization, and static compile time evaluation—in order to support many of Python's language features.

- We show how these modifications can handle a range of Python language constructs that are incompatible with a pure ML-like type system. Among these are Pythonic lambda functions, function passing and returning, decorators, and type queries.

4.1 Localized Type System with Delayed Instantiation

LTS-DI is a bidirectional type system that relies on Hindley-Milner-Damas type inference [57, 88, 44] to determine a type for each expression in a given program ahead of time. The most important parts of this system are *parametric polymorphism* (which provides support for generic types and functions) and *bidirectional type inference* (which allows expression types to be decided after the expression has been processed) [99].

In LTS-DI, each type in a program is either *concrete* (like `int` or `List[int]`), or *generic* (in other words, parameterized by other types, like `List[T]` or `Optional[T]`

```
def foo():
  a = []    # a: List[α]
  b = None # b: Optional[β]
  # At this point, α and β are unbound and
  # cannot be resolved in the scope of foo.
```

```
def foo(x):
  x.append(1)
a = []  # a:       List[α]
foo(a)  # foo(a): Function[[β], η]
        # unify:  β ←List[α]
# At this point, type of a is still unbound α,
# and thus Codon cannot instantiate foo.
```

Figure 4-2: Examples of cases that cannot be type checked by LTS-DI.

where `T` stands for any type). During type inference, LTS-DI maintains a context Γ that maps each type name to known type variables. Initially, Γ is initialized with basic types such as `int`, `float`, `Ptr[T]`, `Generator[T]` and `void`, and grows with each new type and function definition, as well as with variable assignments. The presence of generic types necessitates bidirectional type inference if type annotations are not present, as the exact instantiation of a generic type is often not known at declaration time. For example, an empty list declaration `x = []` has type `List[`τ`]` where τ is a currently unknown—or *unbound*—type variable. τ can later become resolved once we, say, add an element to the list `x`. Resolved types are denoted as *bound* types; examples include concrete types such as `int`, and initially unbound types that have been inferred afterwards. For instance, τ in the previous example will be bound, or *unified*, to `int` by `x.append(1)`. An example of unification is given in Figure 4-1. Ultimately, the goal of type checking is to realize all types in the program and to report any unrealized type as a compiler error.

LTS-DI type system rules take inspiration from the Standard ML type system with let-polymorphism [89, 99], where each let-expression `let x = a in b` is represented by a Pythonic function definition `def x(): a` followed by `b`. However, due to the nature of Python programs and their heavy reliance on duck typing, LTS-DI makes

some novel departures from the Standard ML type system. Most importantly, LTS-DI will not infer the most general type of a generic function in advance through type checking the function (i.e. let-expression) body during definition; instead, it will only use the function signature to obtain the function's type (which, notably, can be more general than the one obtained by a Standard ML approach), and will *delay* inferring types in a function body until the function is instantiated with bound types (i.e. all of its arguments and generics are fully known). With generic functions, each unique combination of function argument types will produce a new *instantiated* function that needs to be type-checked separately. This technique, called *monomorphization*, is used extensively by LTS-DI to instantiate different concrete functions for different argument types (Figure 4-1). The type soundness of the function's body, as well as its return type, will be inferred by the type checking algorithm solely from the provided argument types at the time of realization. If, on the other hand, at least one argument type is not known, LTS-DI will delay—unlike Standard ML—the instantiation (and type checking) of the function, and assign a new unbound type to the function call, hoping that later expressions will fully resolve function arguments and allow for its instantiation, and hence type resolution (note that this approach bears some similarities to Cartesian Product type systems [106]). By combining monomorphization with generic functions (in other words, parametric let-polymorphism) as such, LTS-DI can faithfully simulate Python's runtime duck typing at compile time.

Another important distinction of LTS-DI over an ML-like system is *localization*, which treats each function block as an independent type checking unit with its own typing context Γ. As such, each such block must be locally and independently resolvable by the type system without knowing anything about other blocks. For example, as seen in the top snippet of Figure 4-2, the type of `a` cannot be inferred from the scope of `foo` alone, and as such will produce a compiler error. Because the outermost scope is treated as a function itself, the type of `a` also cannot be inferred from the top-level alone in the bottom snippet of Figure 4-2 (the fact that `foo` can realize `a` does not help, as each function context is independent of other contexts).

However, these two departures—and restrictions at the same time—give LTS-DI much greater flexibility in dealing with generic functions, which are ubiquitous in Python. Most importantly, LTS-DI treats generic functions as *bound types*, and instantiates them as unbound types only during their application. Otherwise, each instance of a function would be treated as unbound and instantiated as soon as possible, losing its genericity after the first instantiation. In other words, if a generic function is passed as an argument to a function `foo`, it can be instantiated differently depending on the supplied arguments within the local context $\Gamma_{\texttt{foo}}$, unlike in ML where the first instantiation determines the type of the function variable. These differences from Standard ML together allow better compatibility with Python, by enabling more general lambda support, as well as support for *returning* generic functions—a necessary requirement for implementing Python's decorators (Figure 4-3).

4.2 Static Evaluation

An ML-like type system such as LTS-DI can sometimes over-eagerly reject a valid Python program. For example, a common Python pattern is to dynamically check the type of an expression via `isinstance`, and to proceed to different blocks of code for different types, often within an `if-else` statement. A similar strategy is also used in conjunction with `hasattr`. However, most type checkers will have to check both branches at the same time and will raise an error if, say, a candidate type does not contain a given method even if the call is guarded by a `hasattr` check. This behaviour stems from the compiler's treating `if`-conditions as purely runtime constructs and assuming that both branches might get executed, resulting in the type checking of both branches in advance. Python, on the other hand, does not distinguish between compile time and runtime and is able to handle these cases gracefully during the latter.

However, both `isinstance` and `hasattr`—as well as many other methods—can in fact generally be resolved at compile time. To this end, LTS-DI borrows from other languages and utilizes the concept of static expressions (akin to `constexpr`s in C++).

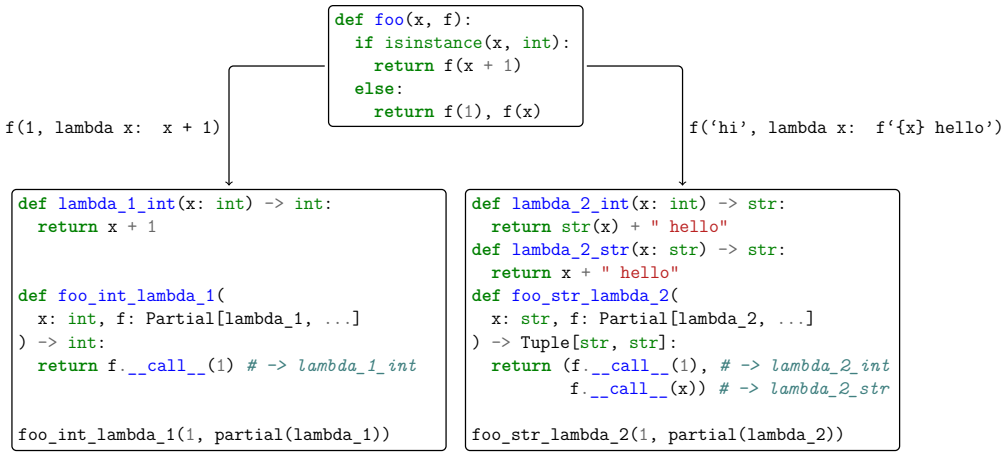


Figure 4-3: Example of monomorphization and static evaluation in LTS-DI. By combining these two, LTS-DI can support many common Pythonic constructs, like `isinstance` type checking, generic functions that return different types on different invocations, and so on.

These expressions are evaluated at compile time, and if an `if`-statement's condition is a static expression, it will be type-checked and compiled *only if* the expression evaluates to true at compile time. Thus the compiler can opt out of compiling blocks that fail a `hasattr` check. An example is provided in Figure 4-3. Note that, unlike many other languages, LTS-DI automatically detects static expressions and does not require the user to manually annotate them.

Static evaluation can be combined with constant expressions that can be used to instantiate types and functions. For example, the n-bit integer type is expressed as `Int[N: int]`, where `N` is not a type generic but a constant expression generic. Constant expression generics behave the same way as other generics—for example, different values of `N` instantiate *different* types—and can be combined with other constant expressions to express fine-grained typing requirements.

4.3 Special Cases

4.3.1 Optional values

In Python, all objects are reference objects (effectively pointers), with no distinction between optional and non-optional references. However, in a strongly-typed system, this distinction is necessary for program validity. To maximize compatibility with Python, LTS-DI automatically coerces non-optionals and optionals to their counterparts in the following cases:

assignment wrapping: (if `var` is an `Optional[T]`)

```
var = non_opt  ⟶  var = Optional(non_opt)
```

function argument wrapping: (if `foo` expects `Optional[T]` as an argument)

```
foo(non_opt)  ⟶  foo(Optional(non_opt))
```

function argument unwrapping: (if `opt` is an `Optional[T]` and `foo` expects `T`)

```
foo(opt)  ⟶  foo(unwrap(opt))
```

access unwrapping: (if `opt` is an `Optional[T]`)

```
opt.member  ⟶  unwrap(opt).member
```

`if`-expression wrapping: (both sides of an `if`-expression must match)

```
non_opt if cond else opt  ⟶  Optional(non_opt) if cond else opt
```

As in Python, implicit or explicit unwrapping of optional types will raise an exception if the given object is `None`.

4.3.2 Function passing

LTS-DI supports partial function creation and manipulation through Python's `partial` construct (in the `functools` module) or via a new internal ellipsis construct (e.g.

`f(42, ...)` denotes a partial function application with the first argument specified). Each partial function is typed as a named tuple of known arguments, where the names correspond to the original function's name. Unlike in ML-like languages, LTS-DI allows functions *and* partial functions to be generic and thus instantiated multiple times differently.

To support lambdas and decorators, LTS-DI automatically "partializes" functions that are passed as an argument or returned as a function value, and as such allows passing and returning generic functions that can be instantiated multiple times. By doing so, the system is able to support decorators that rely on generic function passing and returning.

The cost of allowing generic functions to be passed and returned is that the types of two (partial) functions with the same argument types are often not compatible. In some cases—for example, if both functions are realized and have no partial arguments—LTS-DI can automatically match their corresponding types without error. This approach also results in a somewhat higher number of types and instantiations than a Standard ML-like approach, although duplicate instantiations can be merged later in the compilation pipeline (e.g. LLVM provides a pass to merge identical functions), and thereby have no effect on code size.

Finally, we note that LTS-DI, unlike Python, supports *overloaded methods*. While it is possible to simulate method overloading using static evaluation and `isinstance` checks, it is often cleaner if separate overloads are visually distinct.

4.3.3 Miscellaneous considerations

In order to match the behaviour of Python, LTS-DI processes import statements at *runtime*. This is done by wrapping each import in a function that is called only once by the first statement that reaches it. LTS-DI's LTS-DI also unwraps iterables when needed, and up-casts `int` to `float` when needed. It also has limited support for traits and treats `Callable[...]` and `Generator[...]` as such. Note that LTS-

```
def flatten(v):
  for a in v:
    if hasattr(type(a), "__getitem__"):
      yield from flatten(a)
    else:
      yield a

v = (1, [2, 3], ([4], [5]))
print(list(flatten(v)))  # [1, 2, 3, 4, 5]
```

Figure 4-4: Practical example of LTS-DI involving flattening a nested collection.

DI fully supports generators and coroutines (and, in concert with LLVM, is able to efficiently unroll them to highly efficient loops), exceptions and decorators. Finally, LTS-DI supports Python interoperability and can handle objects managed by the Python runtime via its `pyobj` interface. Such objects are automatically wrapped and unwrapped by LTS-DI, depending on the circumstances, in a similar fashion to `Optional[T]`. As a result, all existing Python modules and libraries (NumPy, SciPy, etc.) are readily usable within LTS-DI.

4.4 Examples

4.4.1 Recursive flatten

As a real-world, non-trivial example, consider the snippet shown in Figure 4-4 implemented in Seq [109] that was modified to use LTS-DI as its type system. The `flatten` function takes an arbitrary collection and recursively *flattens* its contained elements to generate the non-collection elements. Hence, the `print` statement on the last line displays `[1,2,3,4,5,6]`—the inner elements of `v` that are neither lists nor tuples. Executing this function in standard Python is relatively straightforward, as all type information is deferred until runtime. In a statically-typed context, however, the situation is substantially more complicated and requires the combined use of several of LTS-DI's features.

Specifically, for this example, the following steps take place:

1. The type of `v` is deduced based on the right-hand side of the assignment on the penultimate line. This type is T =`Tuple[int,List[int],Tuple[List[int],List[int]]]`.

2. `flatten` is instantiated with argument type T. Since T is a heterogeneous tuple, iteration over it (`for a in v`) is unrolled, and a copy of `a` for each element type of `v` is generated.

3. The first element type is T_1 =`int`. The `hasattr` call is statically evaluated to `False`, so only the body of the `else` statement (`yield a`) is retained. This will yield `1`.

4. The second element type is T_2 =`List[int]`. This time the `hasattr` call is statically evaluated to `True`, so the body of the `if` statement is retained. Therefore, a new instance of `flatten` is created for argument type T_2, whereby this process is recursively repeated. This will yield values `2` and `3`.

5. The third element type is T_3 =`Tuple[List[int], List[int]]`, which again instantiates `flatten` for the new type T_3. This new instantiation will itself reuse the `flatten` instance from the previous step for argument type T_2, or `List[int]`, ultimately yielding values `4` and `5`.

Although this example at first appears out of reach for static type checking, the features employed by LTS-DI—particularly monomorphization and static evaluation—make it tractable.

4.4.2 Dependent collections

As a second example, consider the code in Figure 4-5. Here, the `group` function takes an iterator of key-value pairs and groups them into a dictionary mapping keys to lists of values. The invocation of `group` on the last line instantiates the function for argument type `List[Tuple[str, int]]`; within `group` itself are two collections of unknown types:

```
def group(items):
  groups = {}
  for key,value in items:
    groups.setdefault(key, []).append(value)
  return groups

items = [('a', 3), ('b', 5), ('a', 7)]
print(group(items))  # {a: [3, 7], b: [5]}
```

Figure 4-5: Practical example of LTS-DI involving several collections with unknown, dependent types.

- The `groups` variable of type `Dict[`T_k`,`T_v`]` for unknown key and value types T_k and T_v, respectively.
- The empty list `[]` used in the `setdefault` method call of type `List[`T_e`]` for unknown element type T_e.

LTS-DI then proceeds to deduce these unknown types as follows:

1. The `setdefault` method of the `Dict` type expects argument types K and V, where K and V are the dictionary's key and value types, respectively. Hence, the system deduces that T_k is the type of `key`—or `str` based on the preceding `for`-loop—and that T_v is the type of the empty list—or `List[`T_e`]`.
2. The return value of `setdefault` is the dictionary's value type, which was deduced to be `List[`T_e`]` in the previous step. Further, the `append` method of `List` takes the list's element type as an argument—since `value` is being passed to this method, it is deduced that T_e must be `int`.
3. Given the resolution of T_e in the previous step, it is now concluded that T_v must be `List[int]`. Now, the type of `groups` is fully resolved as `Dict[str, List[int]]` and the type of the empty list as `List[int]`.

This example showcases how LTS-DI's bidirectionality is essential when dealing with collections whose types cannot be deduced immediately. In particular, this snippet contains two collections whose element types are non-trivially intertwined, but which

LTS-DI is able to resolve nonetheless.

4.5 The LTS-DI Algorithm

The LTS-DI type checking algorithm operates on a localized block (or list) of statements that in practice represents either a function body or top-level code (excluding function or class constructs). The crux of LTS-DI's typing algorithm consists of a loop that continually runs the type checking procedure on expressions whose types are still not completely known, until either all types become known or no changes can be made (the latter case implies a type checking error, often due to a lack of type annotations). Multiple iterations are necessary because types of later expressions are often dependent on the types of earlier expressions within a block, due to dynamic instantiation (e.g. `x = []; z = type(x); x.append(1); z()`).

Type checking of literals is straightforward, as the type of a literal is given by the literal itself (e.g. `42` is an `int`, `3.14` is a `float`, etc.). Almost all other expressions—binary operations, member functions, constructors, index operations and so on—are transformed into a call expression that invokes a suitable magic method (e.g. `a + b` becomes `a.__add__(b)`). Each call expression is type checked *only* if all of its arguments are known in advance and fully realized. Once they are realized, the algorithm recursively type checks the body of the function with the given input type argument types, and in practice caches the result for later uses.

Call expression type checking will also attempt to apply expression transformations if an argument type does not match the method signature, an example of which is unwrapping `Optional` arguments. Finally, references to other functions are passed not as realized functions themselves (as we often cannot know the exact realization at the time of calling), but instead as temporary named function types (or partial function types, if needed) that point to the passed function. This temporary type is considered to be "realized" in order to satisfy LTS-DI's requirements.

Below, we provide a formal characterization of the algorithm, and highlight its dif-

ferences with the standard Hindley-Milner type checking algorithm.

4.5.1 Notations and definitions

Before proceeding, we will introduce the following notations and definitions:

- Each function $\mathcal{F}$ is defined to be a list of statements coupled with argument types $\mathcal{F}_1^{\text{arg}}, \ldots, \mathcal{F}_n^{\text{arg}}$ and a return type $\mathcal{F}^{\text{ret}}$.
- Each statement is defined to be a set of expressions $e_1, \ldots, e_m$. Each expression e has a type e_{type}—the goal of type checking is to ascertain these types.
- All types are either *realized* (meaning they are known definitively) or *unrealized* (meaning they are partially or completely unknown), as described in Section 4.1. For example, `int` is a realized type, `List[T]` is only partially realized as `T` is a generic type, and `T` itself is completely unrealized. Let Realized(t) denote whether type t is fully realized.
- Some expressions are *returned* from a function and thus used to infer the function's return type. Let Returned(e) denote whether expression e is returned.
- Let UnrealizedType() return a new, unrealized type instance.
- Unification is the process by which two types are forced to be equivalent. If both types are realized, both must refer to the same concrete type or a type checking error will occur. Partially realized types are recursively unified; for example, unifying `List[T]` and `List[float]` results in the generic type `T` being realized as `float`. Let Unify(t_1, t_2) denote this operation for types t_1 and t_2.
- Define an *expression transformation* to be a function $\xi : \mathbb{E} \mapsto \mathbb{E}$ that converts one expression into another, where $\mathbb{E}$ is the set of expressions. LTS-DI employs a set of expression transformations $\mathcal{X}$ to handle various aspects of Python's syntax and semantics, such as what is described in Section 4.3.

4.5.2 The algorithm

The LTS-DI algorithm is primarily based on two subroutines that recursively call one another. Firstly, LTSDI($\mathcal{F}$) (Algorithm 1) takes a function $\mathcal{F}$ and assigns realized types to each expression contained within, or reports an error if unable to do so. This procedure continually iterates over the contained expressions, attempting to type check each that has an unrealized type. If no expression types are modified during a given iteration, a type checking error is reported. Otherwise, if all expression types are realized, the procedure terminates.

Secondly, TypeCheck(e) performs type checking for the individual expression e. Since this process predominantly entails type checking call-expressions, Algorithm 2 outlines the algorithm specifically for such expressions. Each argument is first recursively type checked, after which the types of the argument expressions are unified with the function's argument types. If unification fails, expression transformations are applied in an effort to reach a successful unification; if none is encountered, an error is reported. At the end, if all argument expression types are realized, the function body is recursively type checked by again invoking LTSDI.

4.5.3 Differences with standard Hindley-Milner inference

The LTS-DI algorithm is based on the Hindley-Milner type inference algorithm, but bears several novel distinctions. Firstly, applications of LTS-DI are *localized* to function bodies: Algorithm 1 takes a function as input, rather than an entire module. Localization enables handling of overloaded functions and methods (absent in most ML-like type systems), which is needed primarily to support Python's magic methods (e.g. `int.__add__(int)` versus `int.__add__(float)`). Secondly, LTS-DI applies the type checking procedure repeatedly so as to account for *delayed instantiation*—unlike ML type systems, which will type check called functions immediately, LTS-DI delays the instantiation and type checking of functions until their argument types are realized (last `if`-statement in Algorithm 2), which in turn emulates Python's duck typing and dynamism at compile time.

Algorithm 1: Type checking of a function $\mathcal{F}$.

Result: LTSDI($\mathcal{F}$)

$\mathcal{F}^{\text{ret}} \leftarrow$ UnrealizedType();
foreach $s \in \mathcal{F}$ **do** // iterate over statements
  foreach $e \in s$ **do** // iterate over expressions
    $e_{\text{type}} \leftarrow$ UnrealizedType();
  end
end
$\mathcal{T} \leftarrow \{(e, e_{\text{type}}) \mid e \in s, \forall s \in \mathcal{F}\}$;
loop
  $\mathcal{T}_0 \leftarrow \mathcal{T}$;
  foreach $s \in \mathcal{F}$ **do** // iterate over statements
    foreach $e \in s$ **do** // iterate over expressions
      if Realized(e_{type}) **then**
        if Returned(e) **then**
          Unify($e_{\text{type}}, \mathcal{F}^{\text{ret}}$);
        end
      else
        $e_{\text{type}} \leftarrow$ TypeCheck(e);
      end
    end
  end
  $\mathcal{T} \leftarrow \{(e, e_{\text{type}}) \mid e \in s, \forall s \in \mathcal{F}\}$;
  if $\bigwedge_{(e,t)\in\mathcal{T}}$ Realized(t) **then**
    return
  else if $\mathcal{T} = \mathcal{T}_0$ **then** // check change in types
    error // type checking error
end

Algorithm 2: Type checking of a *call*-expression $e = (\mathcal{F}, a_1, \ldots, a_n)$ for called function $\mathcal{F}$ and argument expressions $a_1, \ldots, a_n$.

Result: $\text{TypeCheck}_{\text{call}}(e)$

```
foreach a_i ∈ {a_1, ..., a_n} do
    t ← TypeCheck(a_i);
    if ¬Unify(t, F_i^arg) then
        unified ← 0;
        foreach ξ ∈ X do
            a'_i ← ξ(a_i);
            t' ← TypeCheck(a'_i);
            if Unify(t', F_i^arg) then
                unified ← 1;
                break
            end
        end
        if unified = 0 then
            error
        end
    end
end
if ⋀_{a∈{a_1,...,a_n}} Realized(a_type) then
    LTSDI(F);
    return F^ret
end
```

```
(* OCaml *)
let f g a b = (g a, g b) in
  let g x = x in
    f g 1 "a"  (* error *)
```

```
# Python
def f(g, a, b): return g(a), g(b)
def g(x): return x
f(g, 1, "a")  # compiles
```

Figure 4-6: Example of a program that cannot be type checked by standard Hindley-Milner type inference algorithms, but which *can* be type checked by LTS-DI. Since LTS-DI delays function instantiation, it can support multiple applications of the function `f` on different argument types, unlike OCaml.

As a concrete example, consider Figure 4-6. The left snippet shows OCaml code, which is type checked with standard ML type inference; the bottom line produces an error since the type of the passed function `f` is realized by the first function application, leading to inconsistent types on the second application. LTS-DI's delayed instantiation is able to support the equivalent code in Python, since functions are not immediately instantiated.

Finally, LTS-DI employs a number of expression transformations ($\mathcal{X}$ in Algorithm 2) in order to increase Python compatibility. An appealing aspect of this approach is that the set of transformations can be easily expanded to support new semantics and behavior.

4.6 Limitations

The goal of LTS-DI is to support as many features as possible from the original Python feature set without relying on runtime type checking of any kind. As of now, LTS-DI can handle *nearly* everything that Python can, including comprehensions, iterators, generators (both sending and receiving), complex function manipulation, variable arguments via `*args`/`**kwargs`, type checks with `isinstance` and `hasattr`, and more.

However, there are several features that LTS-DI still does not support. Some of the features are deliberately not supported: these include dynamic modifications of types (e.g. dynamic method table modification, dynamic addition of class members,

metaclasses, and class decorators) and dynamic scoping operations (e.g. Python's lax scoping and modification of the internal `__dict__` table). Other unsupported features are, on the other hand, planned to be incorporated into the LTS-DI in the near future. These include support for inheritance and dynamic polymorphism and support for union types.

Further, standard Python modules need to be re-implemented so as to be compatible with LTS-DI, which currently entails implementing some C-based standard modules in Python. In practice, we have found that generally simply copying the "equivalent to" Python snippets from CPython's documentation will simply work without issue. Yet, several Python built-in modules are not yet supported as-is due to this restriction. Many of the features that are not currently supported in LTS-DI are in fact still possible in a statically-typed context. For example, although creating a list containing objects of different types (e.g. `[42, 'foo']`) is disallowed in LTS-DI, it would be possible to support by implicitly converting the list's type to a union type over its element types (e.g. `Union[int,str]` in the preceding example). Several of these features are under active development at the time of writing.

4.7 Conclusion

The type checking methodology presented in this chapter lays the groundwork for Seq's compilation pipeline, and opens the door to novel compilation techniques like *bidirectional compilation*, which we discuss in the next chapter. Statically type checking dynamic languages like Python is a complicated task, and there is still much to explore beyond what is presented here, such as dynamic polymorphism or the use of union types.

Chapter 5

Intermediate Representation

Many languages compile in a relatively direct fashion: source code is parsed into an abstract syntax tree (AST), optimized, and converted into machine code, typically by way of a compiler framework such as LLVM [67]. Although this approach is comparatively easy to implement, ASTs often contain many more types of nodes than necessary to represent a program's underlying computation. This complexity can make implementing optimizations, transformations, and analyses difficult or even impractical. An alternate approach involves converting the AST into an intermediate representation (IR) prior to performing optimization passes. IRs typically contain a substantially reduced set of nodes with well-defined semantics, making them much more conducive to transformations and optimizations.

Seq implements this approach in its IR, which is positioned between the type checking and optimization phases, as shown in Figure 5-1. The Seq Intermediate Representation (SIR) is radically simpler than the AST, with both a simpler structure and fewer nodes. Despite this simplicity, SIR maintains most of the source's semantic information and facilitates "progressive lowering," enabling optimization at multiple layers of abstraction similar to other IRs [68, 127]. Optimizations that are more convenient at a given layer of abstraction are able to proceed before further lowering. A precise listing of SIR's structure can be found in Figure 5-2 and Table 5.1; Appendix D

SIR node	LLVM equivalent	Examples
`Node`	N/A	See below
`Module`	`Module`	N/A
`Type`	`Type`	`IntType`, `FuncType`, `RecordType`
`Var`	`AllocaInst`	`Var`, `Func`
`Func`	`Function`	`BodiedFunc`, `LLVMFunc`, `ExternalFunc`
`Value`	`Value`	See below
`Const`	`Constant`	`IntConst`, `FloatConst`, `StringConst`
`Instr`	`Instruction`	`CallInstr`, `TernaryInstr`, `ThrowInstr`
`Flow`	Various	`IfFlow`, `WhileFlow`, `ForFlow`

Table 5.1: Listing of different kinds of SIR nodes, LLVM IR analogs, and examples.

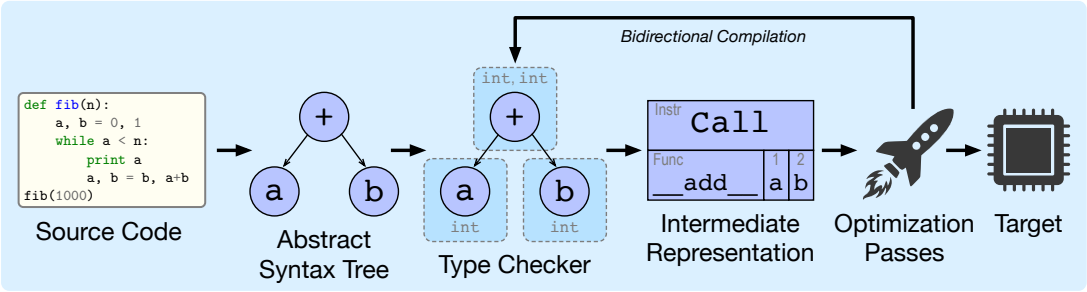


Figure 5-1: Seq's compilation pipeline. Compilation proceeds at first in a linear fashion, where source code is parsed into an abstract syntax tree (AST), on which type checking is performed to generate an intermediate representation (IR). Unlike other compilation frameworks, however, Seq's is *bidirectional*, and IR optimizations can return to the type checking stage to generate new IR nodes and specializations not found in the original program, which is required for several key optimizations we present in Chapters 6 and 7.

provides a much more detailed listing.

Among the contributions of this chapter is the introduction of *bidirectional intermediate representations*, a new class of IRs with which compilation does not follow a linear path after parsing, but can return to the type checking stage during IR passes to generate new specialized IR nodes. The bidirectionality of Seq's IR is critical to many of its general-purpose and domain-specific compiler optimizations.

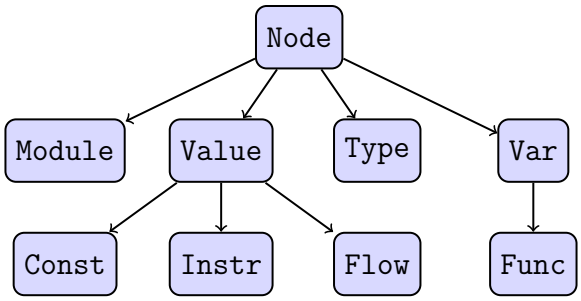


Figure 5-2: Hierarchy of different SIR nodes.

5.1 High-Level Design

SIR is a value-based IR inspired in part by LLVM IR [67]. As in LLVM, we employ a structure similar to single static assignment (SSA) form, making a distinction between *values*, which are assigned once, and *variables*, which are conceptually similar to memory locations and can be modified repeatedly. So as to mirror the source's structure, values can be nested into arbitrarily-large trees. Keeping this SSA-like tree structure enables easy lowering at the Seq IR level. For example, this structure enables SIR to be lowered to a control-flow graph easily.

Unlike LLVM, however, the IR initially represents control flow using explicit nodes called *flows*, allowing for a close structural correspondence with the source code. Representing the control-flow hierarchy explicitly is similar to the approaches taken by Suif [127] and Taichi [58]. Importantly, this makes optimizations and transformations that depend on precise notions of control-flow much easier to implement. One key example is the `for` flow: in Pythonic languages, the `for x in range(y)` pattern is exceedingly common; maintaining explicit loops allows Seq to easily recognize this pattern rather than having to decipher a maze of branches, as is done in lower-level IRs like LLVM IR.

5.2 Operators

SIR does not represent operators like `+`, `-`, etc. explicitly, but instead converts them to function calls resembling Python's "magic methods". For example, the `+` operator

```
@extend
class int:
    @llvm
    def __add__(self, b: int) -> int:
        %tmp = add i64 %self, %b
        ret i64 %tmp
```

Figure 5-3: Primitive operators in Seq via `@llvm` tag.

resolves to an `__add__` call. This enables seamless operator overloading for arbitrary types via these magic methods, the semantics of which are identical to Python's.

A natural question that arises from this approach is how to implement operators for primitive types like `int` and `float`. Seq solves this by allowing inline LLVM IR via the `@llvm` function annotation, which enables all primitive operators to be written in Seq source code. An example for `int.__add__(int)` is shown in Figure 5-3. Information about operator properties like commutativity and associativity can be passed as annotations in the IR.

5.3 Bidirectional Intermediate Representations

Traditional compilation pipelines are linear in their data flow: source code is parsed into an AST, usually converted to an IR, optimized, and finally converted to machine code. With Seq we introduce the concept of a *bidirectional IR*, wherein IR passes are able to return to the type checking stage to generate new IR nodes and specializations not present in the source program. Among the benefits of a bidirectional IR are:

- *Large portions of complex IR optimizations can be implemented in Seq.* For example, the prefetch optimization mentioned in Chapter 3 involves a generic dynamic scheduler of coroutines that is impractical to implement purely in IR.
- *New instantiations of user- or library-defined data types can be generated on demand.* For example, an optimization that requires the use of Seq/Python dictionaries can instantiate the `Dict` type for the appropriate key and value

types. Instantiating types or functions is a non-trivial process that requires a full re-invocation of the type checker due to cascading realizations, specializations and so on.

- *The IR can take full advantage of Seq's intricate type system.* By the same token, IR passes can themselves be generic, using Seq's expressive type system to operate on a variety of types.

While SIR's type system is very similar to Seq's, SIR types are fully realized and have no associated generics (unlike Seq/AST types). However, every SIR type carries a reference to the AST types used to generate it, along with any AST generic type parameters. These associated AST types are used when re-invoking the type checker, and allow SIR types to be queried for their underlying generics, even though generics are not present in the SIR type system (e.g. it is straightforward to obtain the type `T` from a given SIR type representing `List[T]`, and even use it to realize new types or functions).

The ability to instantiate new types during SIR passes is in fact critical to many SIR operations. For example, creating a tuple `(x, y)` from given SIR values `x` and `y` requires instantiating a new tuple type `Tuple[X,Y]` (where the uppercase identifiers indicate types), which in turn requires instantiating new tuple operators for equality and inequality checking, iteration, hashing and so on. Calling back to the type checker makes this a seamless process, however.

Implementing bidirectionality within a given IR requires a degree of integration with the AST and type checker. For example, the type checker employs the host language's type system when type checking the AST, whereas the IR's type system might be significantly different. Seq's IR, for example, has no concept of generic types, whereas generics are used extensively during type checking. To address this issue, all SIR types carry a reference to the corresponding AST type that was used to generate them; this AST type is used when interfacing with the type checker. Furthermore, new IR types are always created via the type checker, ensuring they all carry a corresponding AST

type reference.

We demonstrate the power of bidirectional IRs by implementing several real-world domain-specific optimizations in Chapter 6, each of which relies on bidirectional IR features listed above.

5.4 Seq IR in Action

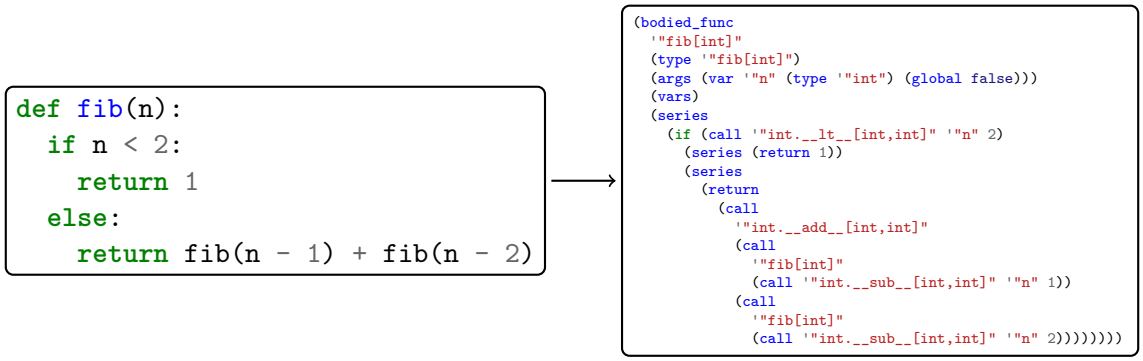


Figure 5-4: Equivalent IR for a simple Fibonacci function in Seq.

Figure 5-4 shows an example of Seq source mapping into SIR. The simple function `fib` (Fibonacci sequence) maps to a SIR `BodiedFunc` with a single integer argument. The body contains an `IfFlow` that either returns a constant or recursively calls the function to obtain the result. Notice that operators like `+` and `-` are converted to function calls (`__add__` and `__sub__`, respectively), but that the IR otherwise mirrors the original source code in its structure, allowing easy pattern matching and transformations.

5.5 Passes and Transformations

SIR provides a comprehensive analysis and transformation infrastructure: users write passes using various SIR builtin utility classes and register them with a `PassManager`, which is responsible for scheduling execution and ensuring that any required analyses are present. In Figure 5-5, we show a simple addition constant folding optimization that utilizes the `OperatorPass` helper, a utility pass that visits each node in an IR

```
class AddFolder : public OperatorPass {
  void handle(CallInstr *v) {
    auto *f = util::getFunc(v->getCallee());
    if (!f || f->getUnmangledName() != "__add__") return;
    auto *lhs = cast<IntConst>(v->front());
    auto *rhs = cast<IntConst>(v->back());
    if (lhs && rhs) {
      auto sum = lhs->getVal() + rhs->getVal();
      v->replaceAll(v->getModule()->getInt(sum));
    }
  }
};
```

Figure 5-5: Simple integer addition constant folder pass in Seq IR. This pass recognizes expressions of the form `<int> + <int>` (where `<int>` is a constant integer) and replaces them with the correct sum.

module automatically. In this case, we simply override the handler for `CallInstr`, check to see if the function matches the criteria for replacement, and perform the action if so (recall that binary operators in SIR are expressed as function calls). Users can also define their own traversal schemes and modify the IR structure at will.

More complex passes can make use of SIR's bidirectionality and re-invoke the type checker to obtain new SIR types, functions, and methods, an example of which is shown in Figure 5-6. In this example, calls of the function `foo` are searched for, and a call to `validate` on `foo`'s argument and its output is inserted after each. As both functions are generic, the type checker is re-invoked to generate three new, unique `validate` instantiations. Instantiating new types and functions requires handling possible specializations and realizing other types and functions (e.g. the `==` operator method `__eq__` must be realized in the process of realizing `validate` in the example), as well as caching realizations for future use to avoid a blowup in code size.

5.6 Code Generation and Execution

Seq uses LLVM to generate native code. The conversion from Seq IR to LLVM IR is generally a straightforward process, following the mappings listed in Table 5.1. Most

```
def foo(x): return x*3 + x

def validate(x, y):
    assert y == x*4

a = foo(10)
b = foo(1.5)
c = foo('a')
```

```
a = foo(10)
validate(10, a)

b = foo(1.5)
validate(1.5, b)

c = foo('a')
validate('a', c)
```

```
class ValidateFoo : public OperatorPass {
  void handle(AssignInstr *v) {
    auto *M = v->getModule();
    auto *var = v->getLhs();
    auto *call = cast<CallInstr>(v->getRhs());
    if (!call) return;
    auto *foo = util::getFunc(call->getCallee());
    if (!foo || foo->getUnmangledName() != "foo") return;
    auto *arg1 = call->front(); // argument of 'foo' call
    auto *arg2 = M->Nr<VarValue>(var); // result of 'foo' call
    auto *validate = M->getOrRealizeFunc("validate",
                         {arg1->getType(), arg2->getType()});
    auto *validateCall = util::call(validate, {arg1, arg2});
    insertAfter(validateCall); // call 'validate' after 'foo'
  }
};
```

Figure 5-6: Example of bidirectional compilation in Seq IR. The simple pass shown in the bottom box searches for calls of function `foo`, and inserts after each a call to `validate`, which takes `foo`'s argument as well as its output and verifies the result. Both functions are generic and can take as an argument any type that can be multiplied by an integer, so the type checker is re-invoked to generate three distinct `validate` instantiations for the example code in the top left box, producing code equivalent to that in the top right box.

Seq types also translate to LLVM IR types intuitively: `int` becomes `i64`, `float` becomes `double`, `bool` becomes `i8` and so on—these conversions also allow for C/C++ interoperability. Tuple types are converted to structure types containing the appropriate element types, which are passed by value (recall that tuples are immutable in Python); this approach for handling tuples allows LLVM to optimize them out entirely in most cases. Reference types like `List`, `Dict` etc. are implemented as dynamically-allocated objects that are passed by reference, which follows Python's semantics for mutable types. Seq handles `None` values by promoting types to `Optional` as necessary; optional types are implemented via a tuple of LLVM's `i1` type and the underlying type, where the former indicates whether the optional contains a value. Optionals on reference types are specialized so as to use a null pointer to indicate a missing value.

Generators are a prevalent language construct in Python; in fact, every `for` loop iterates over a generator (e.g. `for i in range(10)` iterates over the `range(10)` generator). Hence, it is critical that generators in Seq carry no extra overhead, and compile to equivalent code as standard C `for`-loops whenever possible. To this end, Seq uses LLVM coroutines[1] to implement generators. LLVM's coroutine passes elide all coroutine overhead (such as frame allocation) and inline the coroutine iteration whenever the coroutine is created and destroyed in the same function. (We found in testing that the original LLVM coroutine passes—which rely on explicit "create" and "destroy" intrinsics—were too strict when deciding to elide coroutines, so in Seq's LLVM fork this process is replaced with a capture analysis of the coroutine handle, which is able to elide coroutine overhead in nearly all real-world cases.) An example of coroutine usage by the Seq compiler is given in Figure 5-7.

Seq uses a lightweight runtime library when executing code. In particular, the Boehm garbage collector [25]—a drop-in replacement for `malloc`—is used to manage allocated memory, in addition to OpenMP for handling parallelism and `libbacktrace`[2] for exception handling. Seq offers two compilation modes: *debug* and *release*. Debug mode

[1] LLVM coroutines are also used by Clang versions 6 and later to implement the C++ Coroutine Technical Specification [90].

[2] `https://github.com/ianlancetaylor/libbacktrace`

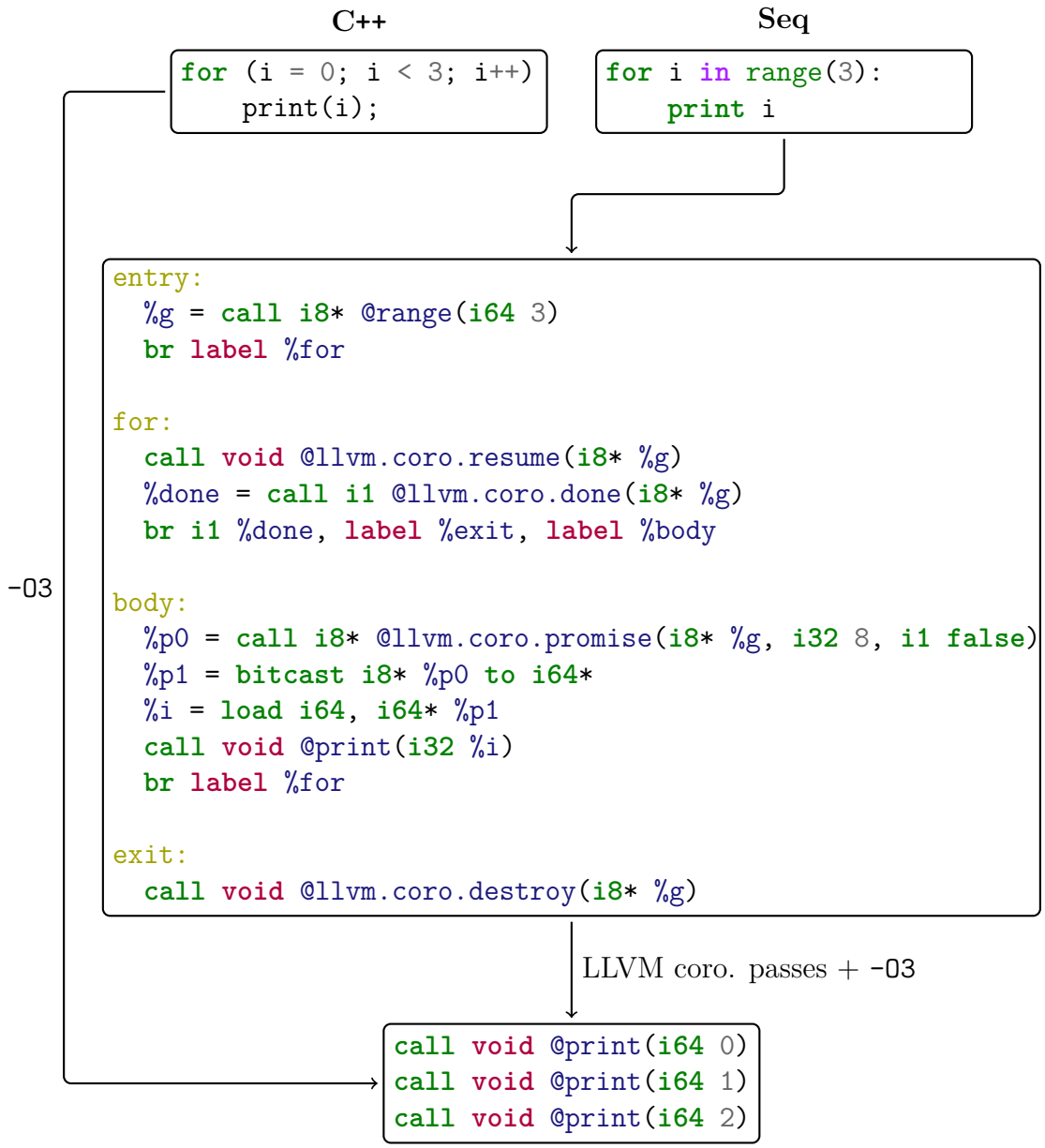


Figure 5-7: Compilation of Seq generators. Two semantically identical loops in C++ and Seq are shown in the uppermost boxes. Seq generators are implemented as LLVM coroutines, iteration over which in LLVM IR is shown in the middle box. The LLVM coroutine passes subsequently deduce that the "range" coroutine is created and destroyed in the same function without escaping, and inline/unroll the coroutine to produce code identical to the C++ example's.

includes full debugging information, allowing Seq programs to be debugged with tools like GDB and LLDB, and also includes full backtrace information with file names and line numbers. Release mode performs a greater number of optimizations (including standard `-O3` optimizations from GCC/Clang) and omits debug information. Users can therefore use debug mode for a seamless programming and debugging cycle, and use release mode for high-performance in deployment.

5.7 Conclusion

Seq's intermediate representation is the medium through which program transformations, optimizations, and analyses are carried out. It offers a well-defined and succinct interface to the primitives that are entailed in many different compiler passes, such as constructing new IR nodes, pattern matching, inlining/outlining, control flow analysis, and so on. Moreover, through bidirectionality, it allows passes to generate, specialize, and instantiate types and functions on the fly, which proves to be immensely useful for many of the optimizations we discuss in the upcoming chapters.

Chapter 6

Genomics-Specific Optimizations

A central goal of Seq is to bridge the gap between the high-performance of low-level languages and the usability of high-level languages. While the usability aspect is largely based on the type system presented in Chapter 4—which allows Seq to reuse Python's syntax—in the next two chapters we focus more on the performance angle. To that end, this chapter describes the many domain-specific optimizations and program transformations that are performed automatically by the Seq compiler, several of which are exceedingly difficult to implement by hand and would require large-scale code changes and significant testing time to incorporate into existing software. Yet, in Seq, they often require just a single additional line of code, resulting in succinct, elegant code that is easy to reason about and maintain. Many of these optimizations also require a global view of the program, and are thus out of reach for software libraries, several of which have been developed for a number of languages [38, 65, 117]. Most libraries are furthermore still bound to high-level host languages that lack performance, or to low-level languages that are hard to program—performance comparisons to some of these libraries are given in Chapter 9.

In short, this chapter provides the following:

- A characterization of Seq's genomic data types and their implementations.

- A host of novel compiler optimizations designed specifically for bioinformatics applications. These include optimizations for reverse complementation, genomic pattern matching, queries of large genomic indices, sequence alignment, and more.

- Several microbenchmarks that demonstrate the efficacy of these optimizations, with in-depth results for larger, real-world applications given in Chapter 9.

6.1 Making Sequences Efficient

It goes without saying that computational genomics applications largely deal with processing and operating on sequences, and so an efficient *sequence type* implementation is invaluable. Seq provides a sequence type `seq` and a separate k-mer type `Kmer[`k`]` for $1 \leq k \leq 1024$. While the sequence type and the k-mer type are conceptually both strings of nucleotide bases, they differ largely in where and how they are used: sequences have arbitrary length and can be accompanied by various metadata like an error profile, quality scores or ambiguous bases; k-mers on the other hand, at least in how they are frequently used in practice, are fixed-size segments of some larger sequence. For example, a typical alignment algorithm would employ short k-mers obtained from an arbitrary-length sequencing read, in conjunction with some k-mer index, to obtain candidate alignments for that read, where k is usually chosen as a fixed parameter beforehand, but read lengths are not known until runtime. In this case, the reads would be of type `seq` and the k-mers would, unsurprisingly, be of type `Kmer[`k`]` for some k. In practice, general sequences can also have a larger alphabet than k-mers (which are restricted to just the four nucleotide bases); for instance, `N` indicates an ambiguous base, `R` indicates an `A` or a `G`, etc. Chapter 3 provides several examples of using sequence and k-mer types in Seq, showing the general syntax as well as some common operations and how to convert between the two.

The following operations on sequences and k-mers are common, and are the subject of several compiler optimizations performed by Seq: *subsequence extraction/iteration*,

reverse complementation, *k-merization* (the process of iterating over a sequence's constituent k-mers), *Hamming distance computation*, *hashing / indexing*, and *dynamic programming alignment* (e.g. Smith-Waterman or Needleman-Wunsch).

6.1.1 Definitions

For a sequence s over an alphabet Σ (where usually $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$), we let $|s|$ be the length of s, $s[i]$ be the character at 0-based index i, $s[i:j]$ be the subsequence from 0-based index i (inclusive) to 0-based index j (exclusive), and $s \parallel t$ be the concatenation of s with another sequence t. Further, let $\overline{s}$ be the *reverse complement* of s. Reverse complementation is a common operation in computational genomics where a given sequence s is transformed into a new sequence t such that $t[i] = \text{RevComp}(s[|s|-i-1])$ for $0 \leq i < |s|$, where

$$\text{RevComp}(c) = \begin{cases} \texttt{T} & c = \texttt{A} \\ \texttt{G} & c = \texttt{C} \\ \texttt{C} & c = \texttt{G} \\ \texttt{A} & c = \texttt{T} \end{cases}.$$

We define a k-mer to be a sequence s of fixed length $|s| = k$. k is typically on the order of 10 to 100. Let $\kappa_k(s)$ be the set of all k-mers that exist as subsequences in sequence s.

These definitions give rise to various algebraic rules, each of which Seq uses to perform various optimizations. For example:

1. $\overline{\overline{s}} = s$: "the reverse complement of the reverse complement is the original sequence" – enables efficient $O(1)$ lazy sequence reverse complementation.

2. $\overline{s} = \overline{s[i:|s|]} \parallel \overline{s[0:i]}, 0 \leq i \leq |s|$: "the reverse complement is the concatenation of the reverse complements of the two halves of the sequence, in reverse order"

– enables efficient k-mer reverse complementation via a lookup table.

3. $\{\bar{t} \mid t \in \kappa_k(s)\} = \{t \mid t \in \kappa_k(\bar{s})\}$: "iterating over reverse complemented k-mers can be done by iterating over k-mers of the reverse complemented sequence" – enables loop reordering for k-mer iteration followed by reverse complementation.

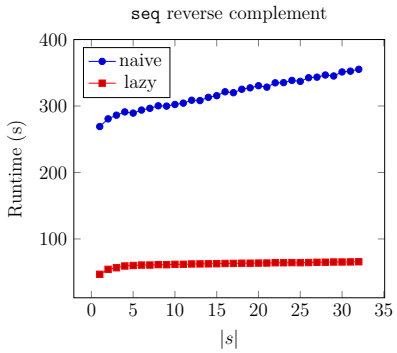
6.1.2 Implementation

Seq's sequence type is implemented as a 16-byte pass-by-value structure containing an 8-byte character pointer and an 8-byte length. The C equivalent would be:

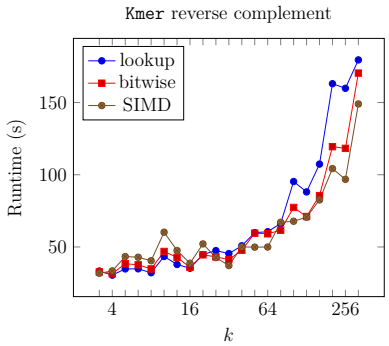
```
struct seq_t { char *ptr; int64_t len; };
```

Subsequence operations do not copy, but rather just return a new sequence instance whose pointer is at the appropriate offset from the original's. Seq then uses a conservative garbage collector to ensure that memory is deallocated as needed, and that nothing is prematurely deallocated, which is especially important when employing this scheme as many pointers will be referring to the middle of some larger allocated block. (In general, there are many ways to implement a sequence data type, and future work entails choosing the best variant based on context at compile time, be it an ASCII representation, a 2-bit encoding or something else. We found the described implementation to work well in a wide range of cases, however.)

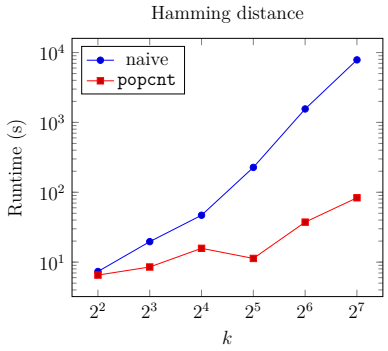
Sequence reverse complementation is an $O(1)$ operation in Seq, implemented not by copying and physically reverse complementing, but by simply, "lazily", flipping the sign of the sequence length. Then, each sequence method (e.g. for subsequence, k-merization, etc.) first checks the sign of the length, and uses it to determine if the sequence should be treated as reverse complemented or not (of course, the actual length is given by the absolute value). Conceptually, therefore, each sequence s is a 2-tuple of the underlying "logical" sequence $s_{\log}$ and a length s_{len}, such that $|s| = |s_{\log}| = |s_{\text{len}}|$, and $s_{\text{len}} < 0$ if and only if s is really reverse complemented with respect to $s_{\log}$. The reverse complementation algorithm is shown in Algorithm 3.



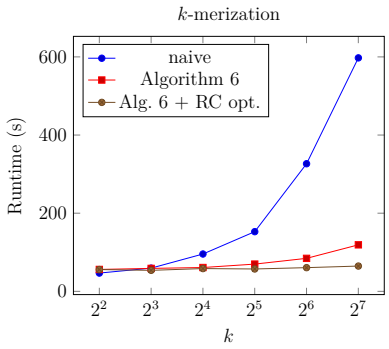
(a) Revcomp opt. (sequence)



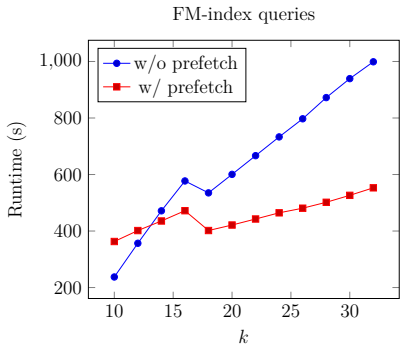
(b) Revcomp opt. (k-mer)



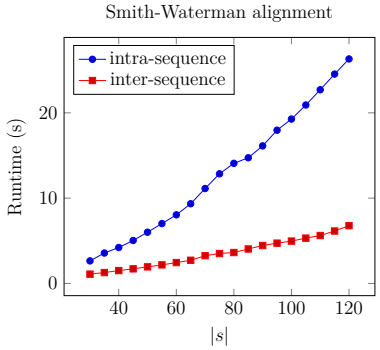
(c) Hamming distance opt.



(d) Pipeline revcomp opt.



(e) Prefetch opt.



(f) Inter-align opt.

Figure 6-1: Effects of several compiler optimizations performed by the Seq compiler on various small benchmarks. Reverse complement performance was measured by counting sequences (or k-mers) lexicographically larger than their reverse complements; Hamming distance performance by computing distances between k-mers and their reverse complements; FM-index query performance by querying k-mers from a set of Illumina reads; Smith-Waterman alignment by aligning one million simulated sequence pairs of maximum length $|s|$ with 90% identity and maximum indel size of 3. The reference genome used in each applicable benchmark was hg19.

Algorithm 3: Reverse complement of sequence $s = (s_{\log}, s_{\text{len}})$.

Result: $\overline{s}$

```
return (s_log, −s_len);
```

An example implementation of $s[\cdot]$ under this scheme is shown in Algorithm 4; notice that the sign of s_{len} must first be checked in order to determine how to index into $s_{\log}$. If s_{len} is positive, we index regularly; if it is negative, on the other hand, we treat $s_{\log}$ as reverse complemented and index correspondingly.

Algorithm 4: Sequence indexing for sequence $s = (s_{\log}, s_{\text{len}})$.

Result: $s[i]$

```
if s_len > 0 then
    return s_log[i];
else
    return RevComp(s_log[(−s_len) − i − 1]);
end
```

The appeal of this approach is that it not only saves memory and copying, but can also be applied multiple times without issue. For instance, clearly $\overline{\overline{s}} = (s_{\log}, -(-s_{\text{len}})) = s$. A more complicated example is $\overline{\overline{s}[i:j]}$, which also works as shown below, along with a concrete example where $s = \texttt{AAAGGGTTTCCC}$ ($|s| = 12$) with $i = 2$ and $j = 5$:

$$\begin{aligned}
\overline{\overline{s}[i:j]} &= \overline{(s_{\log}, -s_{\text{len}})[i:j]} \\
&= \overline{(s_{\log}[|s|-j:|s|-i], -s_{\text{len}})} \\
&= (s_{\log}[|s|-j:|s|-i], s_{\text{len}}).
\end{aligned}$$

$$\begin{aligned}
s &= \texttt{AAAGGGTTTCCC}, \\
\overline{s} &= \texttt{GGGAAACCCTTT}, \\
\overline{s}[2:5] &= \texttt{GAA}, \\
\overline{\overline{s}[2:5]} &= \texttt{TTC}, \\
s[|s|-5:|s|-2] = s[7:10] &= \texttt{TTC}.
\end{aligned}$$

The drawback of this approach is that all sequence operations must include a preliminary check of s_{len}'s sign, although we have found the overhead of this to be minimal, and greatly outweighed by the benefits (Figure 6-1a). In fact, this check can be elided altogether by the compiler in many cases (e.g. when sequences are first read from disk, at which point nothing has yet been reverse complemented).

Figure 6-1a shows the performance of this approach compared to the naive method of simply copying and reverse complementing. The benchmark shown in Figure 6-1a entails counting all subsequences of a given length in the human reference genome that are lexicographically larger than their reverse complement. Notice that the naive implementation is $O(|s|)$ in the best case, whereas the lazy approach is best-case $O(1)$ since bases must only be read until the lexicographic order of the given sequence and its reverse complement can be determined (by contrast, the naive approach must always read all bases of the subsequence to reverse complement it). Hence, the lazy approach not only performs better, but also scales better, as shown in the figure.

6.2 k-mers

In Seq, a k-mer's length is defined by its type (e.g. `Kmer[5]` is a different type than `Kmer[32]`), so unlike sequences the length does not need to be stored explicitly. Seq stores k-mers in a 2-bit encoded format, where `Kmer[`k`]` maps to LLVM IR type `i`N with $N = 2k$. LLVM then maps that integral type to hardware; for $k \leq 32$, a k-mer fits in a single machine register on 64-bit systems, whereas larger k-mers may be spilled to the stack. Longer k-mers can often be stored in MMX/SSE registers, and LLVM will frequently utilize these registers when generating code for long k-mers. Notationally, let $\xi_k(\cdot)$ be the bijection from the set of all k-mers to unique 2-bit encodings.

6.2.1 Reverse complement

The Seq compiler uses three different approaches to reverse complement k-mers (expressed programmatically as `~kmer`):

- *Lookup*: The lookup method uses a lookup table of hard-coded 4-mer reverse complements (taking up $4^4 \times (2 \cdot 4)$ bits, or 256 bytes). Reverse complements of longer k-mers are then constructed using the fact that $\overline{s} = \overline{s[4 : |s|]} \parallel \overline{s[0 : 4]}$ (recursively), i.e. the first four bases are reverse complemented, followed by the

next four and so on, after which they are concatenated (in this case via bitwise shifts) in reverse order. For $k < 4$, the 2-bit encoding can simply be padded with `A` bases to construct a 4-mer, and the resulting `T` bases can be trimmed after reverse complementation.

- *Bitwise*: The bitwise approach uses a series of bitwise operations to reverse the 2-bit pairs of the encoded k-mer, then applies a bitwise-NOT to obtain the complement, the end result of which is the reverse complement. This is a generalization of the approaches used in GATB [46] and Jellyfish [86] for arbitrary-width integer encodings.

- *SIMD*: The SIMD approach uses a vector shuffle instruction to reverse the bytes of the encoded k-mer, then uses bitwise operations on each byte of the reversed vector in a similar fashion as the bitwise approach to obtain the final reverse complement. A similar method is implemented in MMseqs2 [118], although this implementation uses x86-specific `pshufb` instructions to reverse complement individual bytes in the reversed vector, whereas Seq's LLVM IR implementation uses just the target-agnostic LLVM IR `shufflevector` instruction followed by a series of shifts and bitmasks to achieve the same effect.

The performance of each of these approaches for a wide range of k is shown in Figure 6-1b. As shown, there is no clear winner between these three methods. For smaller k (≤ 20), the lookup method is almost always best; for larger k (≥ 32), the SIMD approach is almost always best. For k values between these two cutoffs, the optimal algorithm varies, but we found the bitwise approach to be consistently close to, if not the best option. Because of this, the Seq compiler employs this heuristic to decide how to reverse complement a given k-mer.

6.2.2 k-mer hashing

k-mer hashing is an extensively researched area [83]. For $k \leq 32$, Seq simply uses the encoded 2-bit value as the hash by default. For $k > 32$, on the other hand, the

situation is somewhat more complicated. One option is to use the hash of the first 32 bases, but due to the highly non-uniform structure of the genome, this leads to excessive collisions; for example, all k-mers consisting of a string of 32 `A`s followed by unique bases would hash to the same value. For this reason, hashing k-mers for $k > 32$ is done by hashing the first and last 32 bases, and XORing the two hash values. Compared to the naive approach, this method reduces the average collisions per unique 64-mer in hg19 from 1.089 to 1.007, and more than halves the size of the largest bucket. The best way to hash k-mers can vary based on the application, so Seq enables the user to override the k-mer hash function if needed. Hashing a k-mer in Seq is done via `hash(kmer)`.

6.2.3 Hamming distance

The Hamming distance of two k-mers s and t is defined to be $|\{i \mid s[i] \neq t[i], 0 \leq i < k\}|$; i.e. the number of positions at which corresponding bases differ. In Seq, this is expressed simply as `abs(s - t)`. A naive Hamming distance algorithm is to iterate over the bases of the two k-mers one at a time, and count how many differ. Seq employs a different algorithm using the `popcnt` instruction (which computes the number of 1 bits in an integer's binary representation), shown in Algorithm 5.

Algorithm 5: Hamming distance between two k-mers s and t.

Result: $|s - t|$

$m_1 \leftarrow \overbrace{010101\ldots01_2}^{2k}$;
$m_2 \leftarrow \underbrace{101010\ldots10_2}_{2k}$;
$r_1 \leftarrow (\xi_k(s) \wedge m_1) \oplus (\xi_k(t) \wedge m_1)$;
$r_2 \leftarrow (\xi_k(s) \wedge m_2) \oplus (\xi_k(t) \wedge m_2)$;
$d \leftarrow (r_1 \ll 1) \vee r_2$;
return `popcnt`(d)

Algorithm 5 works by first checking for differences in the odd bits via the m_1 bitmask, then checks the even bits using m_2, and finally ORs the results to obtain a bit-vector with 1s corresponding to different bases, a population count of which is the

desired result. We leave to LLVM the decision of how to lower the logical `popcnt` instruction to actual machine instructions. For longer k-mers, the `popcnt` version can be substantially faster than the naive version, as shown in Figure 6-1c. For example, for 128-mers, the `popcnt` approach is nearly 100× faster.

6.3 Pattern Matching

Seq supports genomic pattern matching via a `match` statement. *Sequence patterns* consist of literal `ACGT` characters, wildcards that match any single base (`_`) and wildcards that match zero or more of any base (`...`), with at most one `...` allowed per pattern. For example, to match a sequence starting with the start codon `ATG` and ending with the stop codon `TAG`, with at least one base in between, one could use the pattern `ATG_...TAG`.

Pattern matching on sequences is done in a straightforward way, where bases are checked one at a time. k-mer pattern matching, on the other hand, is done using bitwise operators. Let $p = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}, \_, ...\}^\star$ be the pattern; we then have the four cases:

1. *No wildcards, no zero-or-more's* ($\_ \notin p, ... \notin p$): This is the simplest case, where the pattern is a literal sequence without wildcards. We simply construct the encoded pattern and compare with the query. We also do a compile time check to ensure $k = |p|$.

2. *Wildcards, no zero-or-more's* ($\_ \in p, ... \notin p$): In this case, we have only single-base wildcards, so we construct a bitmask with zeros at the wildcard positions and ones elsewhere, apply it to both the encoded pattern (with wildcards encoded arbitrarily) and the query, then compare. We again do a compile time check that $k = |p|$.

3. *No wildcards, zero-or-more's* ($\_ \notin p, ... \in p$): Similar to case (1), except we partition the pattern into two sub-patterns p_1 and p_2 around the `...` wildcard,

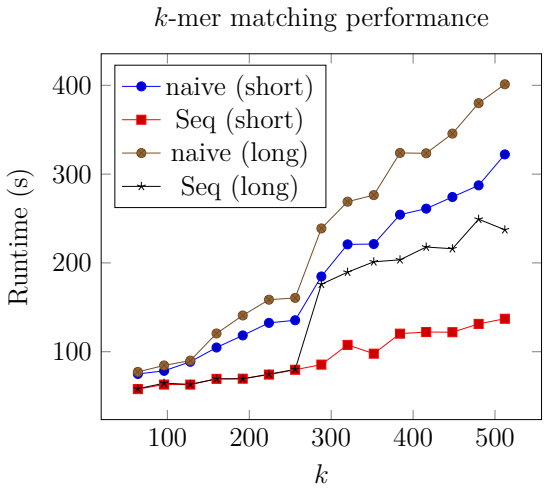


Figure 6-2: k-mer matching performance for a long pattern (`T_` $\times$ 32 $\|$ ... $\|$ `_A` $\times$ 32) and a short pattern (`T_T_..._A_A`). Performance was measured by counting matching k-mers from hg19. The naive implementations consist of two loops that check appropriate bases from the two ends of the k-mer.

then check each individually. This time, we check that $k \geq |p|$ at compile time, since `...` can match zero or more bases.

4. *Wildcards, zero-or-more's* ($\_ \in p, ... \in p$): Similar to (2), except we partition around `...` as above, mask each sub-pattern appropriately, then compare each. We again check that $k \geq |p|$ at compile time.

This scheme is only applied in certain situations: for shorter k-mers and patterns, checking bases one at a time was found to be faster, so the compiler will generate corresponding code in these cases (as a heuristic, we apply bitwise matching only if $k > 256 \vee |p| > 100$. Performance numbers for k-mer matching are shown in Figure 6-2, using both a long and a short pattern for various k.

6.4 Pipelines

Pipelining is a natural model for thinking about processing genomic data. For example, a typical read mapping algorithm can be formulated as the pipeline:

1. Read input reads from FASTQ file

```
from sys import argv
K = Kmer[20]  # seed length
...
index = read_genome_index_from_file(argv[1])
(FASTQ(argv[2])                          |>
 iter                                   ||>
 find_candidate_mappings[K](index)       |>
 filter_candidate_mappings               |>
 smith_waterman_align                    |>
 output_to_disk(argv[3]))
```

Figure 6-3: Hypothetical pipeline for read mapping in Seq, where the first argument (`argv[1]`) is some serialized genome index file, the second is a FASTQ file containing input reads, and the third is the output file. The shown pipeline is parallelized using the parallel pipe operator, `||>`, and each read can be processed in parallel.

2. Extract seeds and lookup in some genomic index
3. Filter candidate alignment positions
4. Perform full Smith-Waterman alignment on remaining candidates
5. Format and output results to SAM file

As shown in Chapter 3, pipelining is supported natively in Seq via the pipe operator, `|>`. Pipelines can be parallelized via the parallel pipe operator, `||>`, which allows all subsequent stages to be executed in parallel (the Seq compiler uses an OpenMP task backend to implement this; future work entails implementing other backends, such as GPU for highly parallelizable tasks or MPI for distributed computing). In Seq, `a |> b` simply passes the output of `a` to `b` if `a` is a function; if `a` is a generator, all values yielded by `a` are passed to `b`. An example of how pipelines might be used to implement a typical read mapping application with the stages described above is shown in Figure 6-3.

The Seq compiler performs various domain-specific optimizations on pipelines, after which they are lowered to a series of (potentially parallel) loops. In the next several sections, we discuss some of these optimizations in detail; some are based on recognizing patterns in the pipeline, whereas others are substantially more involved and

utilize coroutines to obtain better performance. We summarize these optimizations below:

- **Reverse complementation**: Optimizes iteration over k-mers combined with reverse complementation (pattern matching-based).
- **Canonical k-mers**: Optimizes iteration over canonical k-mers (pattern matching-based).
- **Software prefetching**: Optimizes pipelines that access large genomic indices (coroutine-based).
- **Inter-sequence alignment**: Optimizes pipelines that perform sequence alignment (coroutine-based).

6.4.1 k-merization

An extremely common pattern is to iterate over the k-mers of some sequence with a particular step size r. In Seq, this can be achieved using the `kmers[K]` function, where `K` is the target k-mer type. For instance, the following code:

```
K = Kmer[2]
s'ACGT' |> kmers[K](step=1) |> echo
```

prints all 2-mers in the sequence `ACGT` (i.e. `AC`, `CG` and `GT`). Obtaining k-mers with positions as a tuple can be done with the analogous `kmers_with_pos[K]` function.

The best way to perform this k-merization depends on k and the step size r. For example, if $r \geq k$, the best we can do is to simply extract the length-k subsequences at offsets 0, r, $2r$, etc. and convert them to k-mers. However, if $r < k$, then k-mers overlap, so we can reuse the previous k-mer to obtain the next, similar to a rolling hash. The situation is complicated somewhat by the presence of non-`ACGT` bases in the sequence; k-mers spanning such bases are skipped by `kmers`. A simplified version of the k-merization algorithm used in Seq is shown in Algorithm 6. In this algorithm,

the two main branches inside the loop dictate whether the current k-mer must be "refreshed" because a non-`ACGT` base was encountered; if so, the k-mer is re-encoded afresh via SequenceToKmer, otherwise the previous k-mer is "shifted" to the right via ShiftKmer to cover the new bases, without doing a full re-encoding from scratch. Also notice that if $r \geq k$, the second branch is never entered, and we perform a full re-encoding for every k-mer. This style of k-merization is not unique to Seq, and is often used in performance-sensitive genomics applications; the optimizations we discuss below, however, specifically require a domain-specific compiler like Seq to be performed automatically.

Algorithm 6: k-mers contained in sequence s at offsets a multiple of r, skipping non-`ACGT` bases. For simplicity, we ignore reverse complementation and assume s is in the forward direction.

Result: k-mers in s with step size r

$\Sigma \coloneqq \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$;
$i \leftarrow 0$;
$b \leftarrow \texttt{False}$;
$K \leftarrow \underbrace{\texttt{AA}\ldots\texttt{A}}_{k}$;
$\kappa \leftarrow \emptyset$;
while $i + k \leq |s|$ **do**
  if $b = \texttt{True}$ **then**
    $s' \leftarrow s[i : i + k]$;
    if $c \in \Sigma, \forall c \in s'$ **then**
      $K \leftarrow \text{SequenceToKmer}(s', k)$;
      $b \leftarrow r \geq k$;
      $\kappa \leftarrow \kappa \cup \{(i, K)\}$;
    end
  else
    $s' \leftarrow s[i + k - r : i + k]$;
    if $c \in \Sigma, \forall c \in s'$ **then**
      $K \leftarrow \text{ShiftKmer}(K, s')$;
      $\kappa \leftarrow \kappa \cup \{(i, K)\}$;
    else
      $b \leftarrow \texttt{True}$;
    end
  end
  $i \leftarrow i + r$;
end
return κ

6.4.2 Reverse complementation

k-merization followed by reverse complementation can be implemented more efficiently by using property $\{\bar{t} \mid t \in \kappa_k(s)\} = \{t \mid t \in \kappa_k(\bar{s})\}$, as described above. Because of this, the Seq compiler internally makes transformations of the following form:

```
A |> kmers[K](step) |> revcomp |> B
                ↓
A |> _kmers_revcomp[K](step) |> B
```

where `_kmers_revcomp[K]` is a function that iterates over reverse complemented k-mers by first reverse complementing the entire sequence *then* k-merizing.

The effect of these optimizations is shown in Figure 6-1d, which shows the performance of a k-merization and reverse complementation pipeline for a naive implementation (i.e. every k-mer is encoded and reverse complemented individually), an implementation using Algorithm 6 for k-merization, and finally Seq's implementation, which uses both Algorithm 6 and the reverse complement optimization described above.

6.4.3 Canonical k-mers

A "canonical" k-mer is the minimum (often, lexicographic minimum) of a k-mer and its reverse complement. Canonical k-mers are frequently used in k-mer counting algorithms as well as various hashing schemes to ensure that k-mers are treated as equal to their reverse complements [83, 128]. For example, many k-mer counting methods report counts of canonical k-mers rather than raw k-mer counts. Consequently, a common pipeline pattern is to pipe the output of `kmers` to the `canonical` function, which simply returns the canonicalization of its argument.

A naive implementation of such a pipeline might reverse complement each k-mer individually, but a better approach is to use two sliding windows when iterating over k-mers: one for the forward-direction k-mer, and one for the reverse complement. When the step size is 1, for instance, the next base is shifted in on the right of the forward

window, while the *complement* of the base is shifted in on the *left* of the reverse window, thereby ensuring the reverse window always holds the reverse complement of the forward window. Using this approach, no k-mer ever actually needs to be reverse complemented during the iteration. The Seq compiler recognizes this pattern and makes use of a double-sliding-window implementation, `_kmers_canonical`:

```
A |> kmers[K](step) |> canonical |> B
                    ↓
A |> _kmers_canonical[K](step) |> B
```

6.4.4 Software prefetching

Large genomic indices, coupled with cache-unfriendly access patterns, often result in a substantial fraction of stalled memory-bound cycles in many genomics applications [110, 10, 130]. As a result, the Seq compiler performs optimizations to overlap the cache miss latency of an index lookup with other useful work, similar to the general-purpose approaches described in [61] and [35].

To illustrate the effect of this optimization, consider the pipeline `A |> B |> C` where function `B` performs an index lookup. In Seq, by using the `@prefetch` annotation on `B`, the function is implicitly transformed into a coroutine which performs a software prefetch, yields, then commences with the actual index lookup once it is resumed. Code for the pipeline itself is generated to include a dynamic scheduler, which manages multiple instances of the `B` coroutine, suspending and resuming them as needed until all are complete. In this way, while one instance of `B` is performing its prefetch, another can continue doing useful work, compared to the original pipeline where progress would be stalled until a given index lookup completes. A concrete example for FM-indices is shown in Figure 6-4, wherein the typical backwards-extension FM-index search algorithm is applied to count occurrences of subsequences from an input FASTQ file.

The performance of the code from Figure 6-4 for various values of k, both with and without the `@prefetch` line, is shown in Figure 6-1e. For smaller k, the "head" of

```
from sys import argv
from bio.fmindex import FMIndex
fmi = FMIndex(argv[1])
k, step, n = 20, 20, 0
def add(count): n += count

@prefetch
def search(s, fmi):
    intv = fmi.interval(s[-1])
    s = s[:-1]  # trim last base
    while s and intv:
        # backwards-extend intv
        intv = fmi[intv, s[-1]]
        s = s[:-1]  # trim last
    # return count of occurrences
    return len(intv)

FASTQ(argv[2]) |> seqs |> \
 split(k, step) |> search(fmi) |> add
print 'n:', n
```

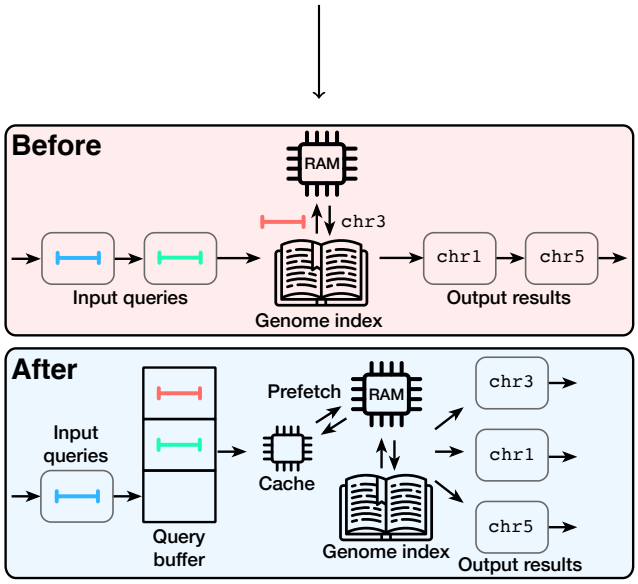


Figure 6-4: Prefetch pipeline optimization in Seq. The code snippet gives an example of using the `@prefetch` annotation with a pipeline performing FM-index queries; the depicted code simply counts occurrences of length-20 subsequences from a given FASTQ using the index. The bottom diagram illustrates the program transformation: pipeline stages that query a large index are converted to coroutines and suspended before querying the index, after issuing a corresponding software prefetch; a dynamic scheduler manages the coroutines and ultimately ensures that any cache miss latency is overlapped with useful work by another coroutine.

the FM-index is largely cache-resident, so the prefetch optimization has no benefit. For larger k, however, the effect of cache misses increases and the optimization significantly reduces runtime (by almost $2\times$ at $k = 32$). The prefetch optimization also works with other types of indices, such as hash tables, although we found it to be most effective on tree-like structures like FM-indices.

An example

A detailed example of the prefetch optimization in action is shown in Figure 6-5. The data structure that is being queried must provide (as the standard data structures from the `bio` module do) a `__prefetch__` magic method definition in the index class, which is logically similar to `__getitem__` (indexing construct) but performs a prefetch instead of actually loading the requested value (and can simply delegate to `__prefetch__` methods of built-in types). The programmer then simply provides a one-line `@prefetch` hint indicating that a software prefetch should be performed.

For instance, an index with a single array `v` may implement `__getitem__(self,x)` by returning `self.v[x]`, in which case it would implement `__prefetch__(self,x)` by returning `self.v.__prefetch__(x)`; i.e. the prefetch is delegated to the underlying array (which may in turn be delegated to a raw pointer, which has intrinsic methods for *actually* performing the prefetch via an LLVM prefetch instruction).

In Figure 6-5, the `@prefetch` annotation first leads to explicit invocations of the `__prefetch__` method, and functions annotated with `@prefetch` (i.e. `process` in the figure) are converted by the compiler into coroutines that yield after each prefetch. Then, pipelines containing such functions as stages are transformed into loops that dynamically schedule multiple invocations of the newly created coroutine, where once one invocation yields or terminates, another is resumed or created by the scheduler, respectively. The transformed pipeline in Figure 6-5, for example, has several noteworthy components:

- `M` is the number of concurrent coroutines to be processed, which ideally should be

__prefetch__ magic method

```
class MyIndex:  # abstract k-mer index
  ...
  def __getitem__(self: MyIndex, kmer: Kmer[20]):
    # standard __getitem__
  def __prefetch__(self: MyIndex, kmer: Kmer[20]):
    # similar to __getitem__, but performs prefetch
```

Function transformations

```
k20 = Kmer[20]
@prefetch
def process(read: seq, index: MyIndex):
  ...
  for kmer in read.kmers[k20](step):
    hits_fwd = index[kmer]
    hits_rev = index[~kmer]
    ...
  return x
```

⟶

```
k20 = Kmer[20]
def process(read: seq, index: MyIndex):
  ...
  for kmer in read.kmers[k20](step):
    index.__prefetch__(kmer)
    index.__prefetch__(~kmer)
    yield
    hits_fwd = index[kmer]
    hits_rev = index[~kmer]
    ...
  yield x
```

Pipeline transformations

```
FASTQ("reads.fq")  # input reads
 |> process(index) # index lookup
 |> postprocess    # output results
```

⟶

```
M = 16   # num. concurrent tasks
N = 0    # next coroutine slot to fill
k = 0    # next coroutine to execute
states = array[generator[T]](M)

for read in FASTQ("reads.fq"):
  if N < M:
    states[N] = process(read, index)
    N += 1
  else:
    while True:
      g = states[k]; g.next()
      if g.done():
        postprocess(g.promise())
        g.destroy()
        states[k] = process(read, index)
        break
      k = (k + 1) % M

for i in range(N):
  g = states[i]
  if not g.done():
    while not g.done(): g.next()
    postprocess(g.promise())
    g.destroy()
```

Figure 6-5: Transformations performed by Seq to enable effective index prefetching. Colored segments under pipeline transformations indicate where the specific stages show up in the resulting code. FASTQ is the standard file format for storing sequencing reads.

large enough to saturate the memory bandwidth of the processor as prefetches are performed. In practice, we choose `M` conservatively to be 16, which also allows for software prefetching performed by other parts of the system, such as the garbage collector.

- `N` is a variable indicating how many of the `M` coroutine slots have been filled, and is only used at the start of the loop to actually fill the slots (initially zero).
- `k` is the next coroutine slot to be resumed by the loop (initially zero).
- `states` is the array holding the `M` coroutine handles/frames (which have type `generator` in Seq). In reality this array is stack-allocated in the entry block of the function containing the pipeline. (`T` is simply the original return type of `process`.)

The code generated in the loop body is that of a simple dynamic scheduler where:

- The `if N < M` component initially populates the array of pending coroutines `states`.
- Inside the `else` clause is a loop that iterates cyclically through `states` and resumes each coroutine. If a coroutine terminates (i.e. `if g.done()`), then the value returned by the coroutine (given by `g.promise()`) is sent through the remainder of the pipeline, as it would be in the original untransformed pipeline; then, the coroutine is destroyed and a new one is created to take its place.
- The final loop simply completes any remaining coroutines that have not yet terminated. Since the number of such coroutines is at most `M`, this loop just executes them sequentially.

By employing this scheme, the latency of one coroutine's cache miss can be overlapped with useful work from another, increasing memory-level parallelism and overall throughput. Note that these optimizations depend only on the existence of a prefetch instruction, which is the case for nearly any modern architecture.

Implementation in Seq IR

The actual prefetch optimization is implemented as a Seq IR pass, in roughly 100 lines of code. The two necessary transformations for this optimization are 1) converting the annotated function to a coroutine and 2) inserting the dynamic scheduler in the pipeline. The former is shown in Figure 6-6 and the latter in Figure 6-7. Due to Seq IR's bidirectionality, the entire scheduler can be implemented in Seq as a generic function, and instantiated during the IR pass by re-invoking the type checker.

6.4.5 Inter-sequence alignment

Sequence alignment via a dynamic programming algorithm like Smith-Waterman is arguably one of the most prevalent operations in genomics applications, and has been the subject of much research and optimization, particularly with regards to SIMD parallelization [120, 43, 104]. There are two approaches to SIMD alignment, referred to as *intra-sequence* and *inter-sequence*. An intra-sequence implementation uses SIMD to align a single pair of sequences, whereas an inter-sequence approach aligns a batch of pairs at once, and uses SIMD across different sequence pairs rather than within one pair. Inter-sequence alignment can be substantially faster than intra-sequence, but is rarely used in practice due to programming difficulty [122].

In Seq, however, inter-sequence alignment is just as easy to program as intra-sequence, as shown in Figure 6-8. When a function marked `@inter_align` is used in a pipeline, the Seq compiler will perform transformations similar to those used for software prefetching; specifically, the function is transformed into a coroutine and suspended just before the alignment. During execution of the pipeline, several hundred suspended coroutines are batched, have their sequences aligned using inter-sequence alignment, and are resumed after the alignment result is returned to them. Similar to the prefetch optimization, this is orchestrated by a dynamic scheduler of coroutines, the code for which is generated in the body of the pipeline. An illustration of such a pipeline before and after this transformation is shown in Figure 6-8.

```
class PrefetchFunctionTransformer : public Operator {
  // return x --> yield x
  void handle(ReturnInstr *x) override {
    auto *M = x->getModule();
    x->replaceAll(
      M->Nr<YieldInstr>(x->getValue(), /*final=*/true));
  }

  // idx[key] --> idx.__prefetch__(key); yield; idx[key]
  void handle(CallInstr *x) override {
    auto *func =
      cast<BodiedFunc>(util::getFunc(x->getCallee()));
    if (!func ||
        func->getUnmangledName() != "__getitem__" ||
        x->numArgs() != 2) return;

    auto *M = x->getModule();
    Value *self = x->front(), *key = x->back();
    types::Type *selfType = self->getType();
    types::Type *keyType = key->getType();
    Func *prefetchFunc = M->getOrRealizeMethod(
      selfType, "__prefetch__", {selfType, keyType});
    if (!prefetchFunc) return;

    Value *prefetch = util::call(prefetchFunc, {self, key});
    auto *yield = M->Nr<YieldInstr>();
    auto *replacement = util::series(prefetch, yield);

    util::CloneVisitor cv(M);
    auto *clone = cv.clone(x);
    see(clone); // don't visit clone
    x->replaceAll(M->Nr<FlowInstr>(replacement, clone));
  }
};
```

Figure 6-6: Function-to-coroutine transformer for Seq IR. This transformation is utilized by several of Seq's pipeline optimizations, including the prefetch optimization.

```
@inline
def _dynamic_coroutine_scheduler[A,B,T,C](
    value: A, coro: B, states: Array[Generator[T]],
    I: Ptr[int], N: Ptr[int], M: int, args: C):
    n = N[0]
    if n < M:
        states[n] = coro(value, *args)
        N[0] = n + 1
    else:
        i = I[0]
        while True:
            g = states[i]
            if g.done():
                if not isinstance(T, void):
                    yield g.next()
                g.destroy()
                states[i] = coro(value, *args)
                break
            i = (i + 1) & (M - 1)
        I[0] = i
```

Figure 6-7: Coroutine scheduler for Seq's prefetch optimization. A pipeline stage marked with `@prefetch` is converted to a coroutine by the pass in Figure 6-6, and this scheduler is used to manage multiple instances of this coroutine, overlapping cache miss latency from one with useful work from another. Scheduler state is passed as pointers (`Ptr[int]`) and modified by the scheduler, which itself gets inserted in the pipeline. Seq IR's bidirectionality is used to instantiate the scheduler with concrete argument types when applying the prefetch optimization.

```
from sys import argv
in1 = argv[1] + '.1.txt'
in2 = argv[1] + '.2.txt'

@inter_align
def process(t):
    query, target = t
    # align and get score
    score = query.align(target).score
    print query, target, score

zip(seqs(in1), seqs(in2)) |> process
```

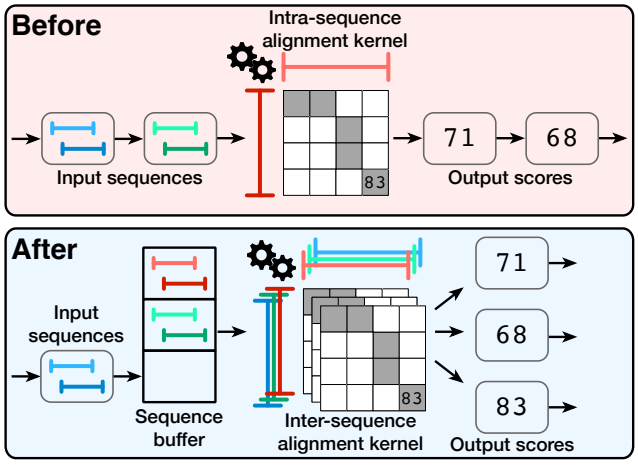


Figure 6-8: Inter-sequence alignment pipeline optimization in Seq. The code snippet gives an example of inter-sequence alignment using the `@inter_align` annotation, where sequence pairs are read from two text files, aligned, and printed. The bottom diagram illustrates the program transformation: in the original pipeline, sequence pairs are passed through the pipeline one at a time and aligned individually; in the transformed pipeline, sequence pairs are batched by a dynamic scheduler, aligned with inter-sequence alignment, and returned to their respective suspended coroutines.

Performance of inter-sequence alignment compared to intra-sequence is shown in Figure 6-1f, where inter-sequence is shown to be almost 4× faster in some cases. The code difference between the two implementations used in this figure is just a single line: the `@inter_align` annotation on the function performing the alignment. Note that Seq uses KSW2 [74, 120] as its intra-sequence alignment kernel.

This approach to alignment also lends itself well to other backends. For example, it would be possible to instead perform inter-sequence alignment on a GPU or FPGA rather than a CPU, and tune the parameters (e.g. sequence buffer size) for each backend. A domain-specific language and compiler such as Seq makes it possible to explore various backends and their benefits in a systematic way, and we plan to pursue this as future work.

6.5 Conclusion

Compiler design and computational genomics have traditionally been isolated from one another, with very little research taking place at their intersection. A key takeaway of this chapter is that there is much to be gained by looking at bioinformatics and genomics through the lens of compilers, particularly when it comes to performance. There are many more genomics-specific compiler optimizations that are worthy of exploration; for example, can we determine the best representation of a sequence based on context? Or, can we take advantage of the highly non-uniform structure of the genome? These questions lead to many interesting avenues for future work.

Chapter 7

Other Optimizations

Beyond genomics-specific optimizations, Seq's Pythonic roots also give rise to many unique optimization opportunities. Specifically, many prevalent idioms and code patterns in Python are particularly inefficient if compiled naively. This chapter introduces several Python-specific compiler optimizations to address this issue, which further liberates the programmer to write straightforward, idiomatic code while leaving all performance considerations to the compiler. Later in the chapter we discuss several general-purpose optimizations and analyses performed by the compiler.

7.1 Python-Specific Optimizations

Given Seq's roots in Python, it is able to substantially accelerate many standard Python programs out of the box. Further, Seq IR makes it easy to optimize several patterns commonly found in Python code. Below we discuss two examples: the dictionary get/set optimization for updating Python dictionaries, and intermediate string optimizations for eliding the creation of unnecessary intermediate strings.

7.1.1 Dictionary get/set optimization

For many applications, dictionary queries and modifications account for a large frac-

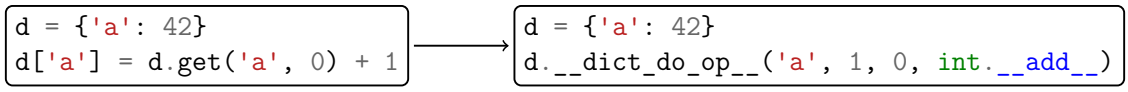


Figure 7-1: Example of dictionary optimization execution. The pass recognizes the *get/set* pattern and replaces it with a single call to `__dict_do_op__`. As this function is generic, we instantiate a new version and pass the `int.__add__` function as a parameter.

tion of the runtime, particularly for "counting" tasks. In Pythonic code, however, these can be a major source of inefficiency. Specifically, we identify the *get/set* pattern, which includes statements of the form `d[x] = d.get(x, <value>) <op> <value>`, as a particularly inefficient idiom. Without optimization, these unnecessarily perform two separate dictionary lookups: one to obtain the value for a key and another to assign a new value. However, we can replace this pattern with a single call that applies the modifications in-place. In Figure 7-1, we show an example of this optimization running on a simple snippet. In implementing this transformation, we showcase the utility of Seq's bidirectional IR—rather than manually implementing the in-place operation in IR, we simply instantiate a helper method. This has the benefit of generalizing the optimization to all types that implement the appropriate methods. The pass that performs this optimization searches for index assignments of the form `d[k] = d.get(k, i) + j` for arbitrary `d`, `i`, `j`, `k`.

7.1.2 Intermediate string optimizations

We additionally implement a simple transformation to "fold" consecutive string additions into one concatenation call. This enables Seq to easily reduce the string allocation overhead of large addition trees. Given that format strings ("f-strings") in Python/Seq de-sugar into concatenation as well, outputting these results using `print` or `file.write` is especially common. We denote these occurrences as the *concatenation/output* pattern. As with *get/set*, this is easy to recognize in the IR and equally simple to optimize. To reduce intermediate values, we simply realize a new version of the appropriate I/O function using the concatenation arguments. Figure 7-2 demonstrates a simple instantiation of this procedure, applied to `print`

```
class PrintOptimization : public OperatorPass {
  void handle(CallInstr *v) {
    auto *M = v->getModule();
    if (auto *callee = util::getStdlibFunc(v->getCallee(),
                                           "print")) {
      auto *internalTuple = extractCatArguments(v->front());
      if (!internalTuple)
        return;

      vector<types::Type *> types = getArgTypes(callee);
      vector<Value *> args = cloneArgs(callee);
      types[0] = internalTuple->getType();
      args[0] = internalTuple;
      auto *print = M->getOrRealizeFunc("print", types, {},
                                        "std.internal.builtin");
      v->replaceAll(util::call(print, args));
    }
  }
};
```

Figure 7-2: Simplified implementation of the *concatenation/output* optimization for `print`. This pass recognizes the pattern, extracts the `cat` function's arguments, clones them, and uses them to realize a `print` function that directly outputs the string rather than calculating an intermediate value.

calls.

7.2 General-Purpose Optimizations

Seq also performs numerous standard, general-purpose optimizations and analyses at the IR level. These include:

- Constant folding and propagation
- Dead code elimination
- Canonicalization
- Flow and dominator analysis

Worth noting is the fact that LLVM will itself apply many of these optimizations

prior to final code generation; hence, the two primary reasons for incorporating them at the Seq IR level are:

- To account for higher-level constructs that may get lost at the LLVM level. For example, Seq IR is able to fold the concatenation of two constant `List`s into one value, whereas recognizing such a pattern at the LLVM IR level is much more difficult.

- To extend the reach of other optimizations. For example, the inter-sequence alignment optimization discussed above requires the alignment parameters to be constant (or global); the constant folding and propagation IR passes allow for cases where the parameters are not explicitly constant in the original program, but are found to be constant after applying propagation/folding.

Below we discuss these passes and optimizations, as well as some of the infrastructure surrounding them.

7.2.1 Analyses

Seq provides several analyses based on the notion of control-flow graphs.

Control-flow graphs

While SIR's hierarchical structure is helpful for pattern recognition, it can occasionally be a hindrance to certain other analyses, particularly those involving data flow. Consequently, we define an auxiliary deconstructed version of SIR. This approach, akin to that taken by Taichi [58], enables analyses to operate over a fully simplified structure without eliminating SIR's nested structure too early.

As in the classic formulation, we define "basic blocks" that contain values, and edges between these blocks [39]. A standard `if-else` flow, for example, will be deconstructed into two blocks each leading to an "end" block. This structure is particularly useful because each value in a given block is guaranteed to execute without interrup-

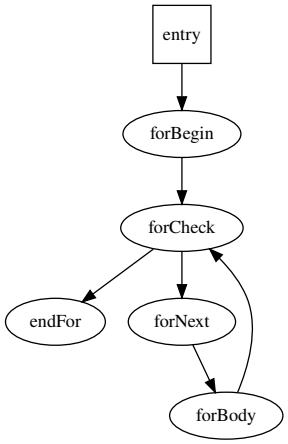


Figure 7-3: Example `for` loop control-flow graph.

tion. Therefore, analyses can often operate on blocks as a whole without having to consider individual instructions therein.

In Figure 7-3, we show the control-flow of a Seq function containing a single `for`-loop. Seq defines control-flow graph conversion methods for all builtin nodes.

Reaching definitions

Given a CFG, it is often useful to calculate the possible values, or definitions, of variables at a particular stage in a program [39]. We implement these analyses by determining the variable definitions *generated* (*gen*) and *killed* (*kill*) by each block. Subsequently, we repeatedly calculate for each block b, $in_b = \bigcup_{i \in preds_b} out_i$ and $out_b = (in_b - kill_b) \cup gen_b$. Once these sets stabilize, the analysis is complete and the reaching definitions can be easily propagated to each individual instruction.

Dominators

In a control-flow graph, a particular node d *dominates* another node v if all paths from the entry to v must go through d. Since this information can be useful for many optimizations, we add support for this calculation as a standard analysis pass. We use a simple approach, similar to that of reaching definitions: for each basic block b, we repeatedly calculate, $dom_b = \{b\} \cup (\bigcap_{i \in preds_b} dom_i)$ until the sets stabilize. While this

```
(for (call '"range.__iter__[range]"
       (call '"range.__new__[int]" 10))
  (var '"i")
  (series)
)
```

Figure 7-4: Simplified SIR equivalent of a `for`-loop.

```
struct RangeMeta {
  bool valid;
  Value *start;
  Value *end;
  int64_t step;
};

void ImperativeForFlowLowering::handle(ForFlow *v) {
  auto *M = v->getModule();

  auto r = analyzeRange(v->getIter());
  if (!r.valid || r.step == 0)
    return;

  v->replaceAll(M->N<ImperativeForFlow>(v->getSrcInfo(),
                                        r.start,
                                        r.step,
                                        r.end,
                                        v->getBody(),
                                        v->getVar()));
}
```

Figure 7-5: Simplified C++ implementation of `for` loop lowering.

approach is simple both conceptually and in its implementation, there are dominator tree construction algorithms with better asymptotic complexity in theory [71, 40]. However, since this analysis is applied at the function level, we found the simple algorithm to be more than sufficient in practice.

7.2.2 Passes

Loop lowering

Seq's canonical loop format can be cumbersome for some optimizations. Consider the

simple loop `for i in range(10)`, which maps to the SIR in Figure 7-4. The loop condition, implicitly $i < 10$, is hidden under two magic method calls. Since this is an extremely common case, we add the `ImperativeForFlow` node, which represents a simple C-like loop, and a simple lowering pass shown in Figure 7-5. The pass simply checks for a `__iter__` call applied to a `range`, and extracts the arguments to create a new loop.

This is an example of "progressive lowering" [68]: the original loop form could be useful to certain passes, while the lowered form may be more helpful for others. In particular, the `ImperativeForFlow` is generally more helpful for loop unrolling, tiling, and vectorization.

Code simplification

Given the reaching definitions analysis, it is possible to determine that a variable's value at a particular usage is a constant. To take advantage of this knowledge, we incorporate a propagation optimization that replaces all such variables with their constant value. We show its implementation in Figure 7-6: for each `VarValue`, we simply check if it has a single, constant reaching definition. If so, we can safely replace the old value with the constant.

This optimization results in a considerable number of arithmetic operations containing only constants. We optimize this case with a folding/simplification pass that runs after constant propagation. Similarly, we resolve control flow (i.e. `IfFlow`s and `TernaryInstr`) at compile time for constant conditions, so as to eliminate dead code.

Canonicalization

There are many different ways to express the same operation in high-level languages—for example, `a + 1` and `1 + a` are syntactically distinct expressions in Seq, but semantically they are equivalent. As a result, passes and analyses that search for specific patterns in SIR would be required to check many permutations that are semantically identical. To remedy this issue, Seq applies a canonicalization pass that converts these

```cpp
class ConstPropPass : public OperatorPass {
private:
  std::string reachingDefKey;
public:
  // boilerplate and okConst not included
  void handle(VarValue *v) override {
    auto *r = getAnalysisResult<RDResult>(reachingDefKey);
    if (!r)
      return;

    auto *c = r->cfgResult;
    auto it = r->results.find(getParentFunc()->getId());
    auto it2 = c->graphs.find(getParentFunc()->getId());
    if (it == r->results.end() || it2 == c->graphs.end())
      return;

    auto *rd = it->second.get();
    auto *cfg = it2->second.get();
    auto reaching = rd->getReachingDefinitions(v->getVar(), v);

    if (reaching.size() != 1)
      return;
    auto def = *reaching.begin();
    if (def == -1)
      return;

    auto *constDef = cast<Const>(cfg->getValue(def));
    if (!constDef || !okConst(constDef))
      return;

    util::CloneVisitor cv(v->getModule());
    v->replaceAll(cv.clone(constDef));
  }
};
```

Figure 7-6: Simplified C++ implementation of constant propagation.

different, semantically equivalent constructs into a single, "canonical" representation.

The central concept used by this pass is that of "ranking" an IR node. Then, for instance, commutative binary operators like addition or multiplication can have their operands re-ordered based on rank. Nodes are ranked based on the following criteria, in order of importance:

- Whether the node is constant (constants come last in ranking)
- Maximum node depth (shallower nodes come last in ranking)
- Node hash, based on types and variables used by the node and its children

Then, the following transformations are performed:

- Chains of commutative/associative operators are re-ordered by rank. For example, `c + b + a` would be re-ordered as `a + b + c`.
- Comparisons are re-ordered by rank. For example, `b < a` would be re-ordered as `a > b`.
- Variables are factored out of addition-multiplication expressions. For example, `a*x + b*x` would be re-written as `(a + b) * x`.
- Constant subtractions are converted into additions. For example, `a - 1` would be re-written as `a + -1`.

Collectively, these rules ensure that most expressions only have a single canonical representation in the IR. Care must also be taken to apply these transformations only when the operand types allow for it—for example, string concatenation via the `+` operator is not commutative, and floating point arithmetic is in general neither associative nor distributive. Because of this, Seq uses the annotations `@commutative`, `@associative`, and `@distributive` to mark the operators on which canonicalization can be applied. Core types like `int` and `float` use these annotations where appropriate, and they can be used in user-defined classes to enable further canonicalization.

7.3 Conclusion

The overarching goal of the compiler passes presented in this chapter is to provide a comprehensive ecosystem of optimizations and analyses, so as to bolster the effect of other domain-specific passes. Operations such as control flow analysis, dominator analysis, constant propagation, and others are all useful in domain-specific settings, and in fact interoperate with Seq's domain-specific optimizations.

Chapter 8

Beyond Genomics

Beyond just genomics and bioinformatics, domain-specific languages are able to provide intuitive, high-level abstractions that domain-experts can easily work with, all the while attaining better performance than general-purpose languages through domain-specific compiler optimizations. Yet, implementing new DSLs is a burdensome task, leading to DSL designers' often embedding them in general-purpose languages. While low-level host languages like C/C++ perform far better than high-level languages like Python, high-level languages are nowadays much more prevalent amongst practitioners in many domains. Here, we generalize Seq to a domain-extensible compiler framework called Codon, which similarly offers Python's syntax, semantics and libraries with zero runtime overhead, achieving the performance of C/C++. We also showcase other DSLs built with Codon for various domains.

8.1 Designing Domain-Specific Languages

Domain-specific languages usually come in one of two varieties: *embedded* or *standalone*. Embedded DSLs are integrated with a general-purpose host language, whereas standalone DSLs introduce new languages, syntax, and semantics. Both varieties have pros and cons, but in practice embedded DSLs are far more common due to ease of

implementation and adoption. Consequently, designers of high-performance DSLs face a crucial choice at the onset: which language to embed their DSL in. In the past, the choice was obvious—high performance required embedding in a lower-level, statically analyzable language like C or C++. Since these languages were already commonly used in their respective domains, such DSLs frequently saw seamless adoption with few issues. Further, by virtue of their host languages, embedded DSLs inherited existing language and compiler infrastructure, providing overall end-to-end performance and general-purpose functionality. Indeed, this approach was taken by many high-performance DSLs like Halide [101], Taco [62] and Tiramisu [13], which each derived from C++—the go-to language at the time for their target audiences.

Nowadays, the decision is no longer so simple. Dynamic languages like Python and Ruby, combined with the widespread availability of relatively high-performance domain-specific libraries [55, 97, 2, 95], have captured the majority of potential users targeted by DSLs. Thus, building new DSLs on top of lower-level languages can in fact become a barrier to widespread adoption, and even alienate a large fraction of the potential user base—indeed, this was Seq's motivation for using Python. Nonetheless, building optimized language features on top of intrinsically low-performance languages like Python or Ruby can be perilous, resulting in tools like TensorFlow [2] or PyTorch [95] primarily using their host languages as APIs for interacting with optimized C/C++ libraries. This approach is only possible because the work performed in Python is minimal, and the heavy lifting is actually done by the C/C++ libraries. Moreover, domains like genomics often make this approach infeasible, particularly when the data comprise billions of small objects directly defined and accessed by the DSL, drastically increasing the cost of interfacing with an external library [109].

The emerging issue, therefore, in today's DSL landscape is that practitioners are increasingly moving towards dynamic languages like Python and R, whereas high-performance DSLs necessitate a foundation in low-level languages like C or C++. Ideally, we would like to combine the familiarity and usability of dynamic languages with the high-performance of low-level languages—much like what Seq has done in

the genomics field—albeit in a domain-agnostic way.

In this chapter, we introduce Codon, a novel solution to bring high-performance DSLs to the Python user community, by building a flexible development framework on top of Seq's optimized, Pythonic base. Codon is a full language and compiler that—like Seq—borrows Python's syntax, semantics, and library features, but compiles to native machine code with zero runtime overhead, allowing it to rival C/C++ in performance. To this end, Codon leverages many of Seq's features: a specialized bidirectional type checking algorithm and novel intermediate representation (IR) to enable optional domain-specific extensions—both in the language's syntax (front-end) and in compiler optimizations (back-end)—via a plugin system that not only allows new DSLs to be seamlessly built on top of the framework, but that also enables different DSLs to be composed within a single program.

Overall, this chapter makes the following contributions:

- We generalize Seq's compiler infrastructure by designing a plugin-based *domain-extensible* compiler framework, where even Seq itself can be realized as a plugin.
- We show how Seq's front-end can be extended with new keywords, and how Seq IR can be extended with new IR nodes, types, optimizations, and analyses.
- We show various DSLs built with this new framework.

8.2 A Domain-Extensible Compiler

Due to the framework's flexibility and bidirectional IR, as well as the overall expressiveness of Python's syntax, a large fraction of the DSL implementation effort can be deferred to the Seq source. Indeed, as shown in Chapter 6, a large fractions of Seq's domain-specific components are implemented in Seq itself, and the same philosophy is carried over to Codon. This has the benefit of making Codon DSLs intrinsically interoperable—so long as their standard libraries compile, disparate DSLs can be used together seamlessly. Along these lines, Codon uses a modular approach for in-

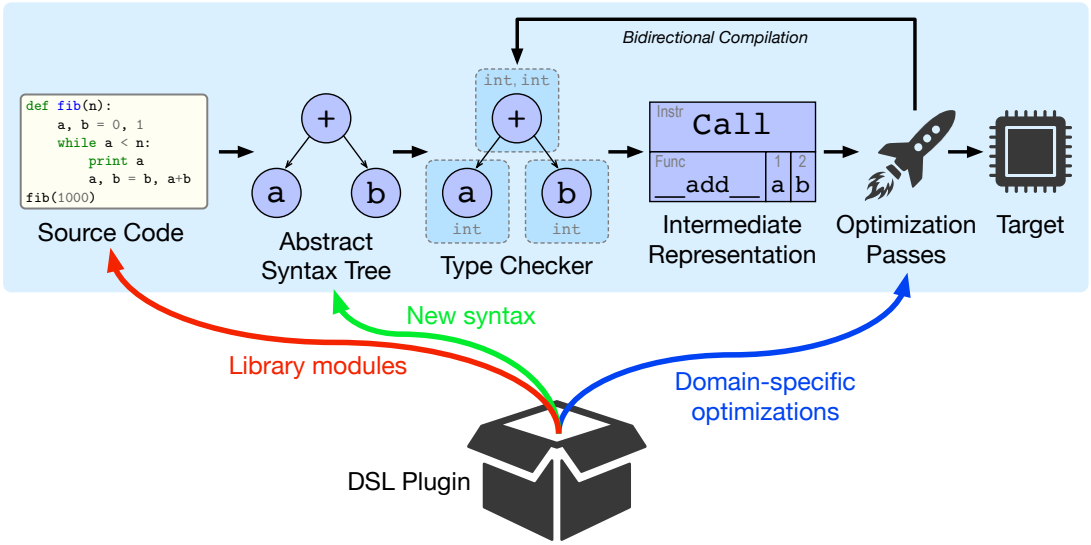


Figure 8-1: Codon's compilation pipeline. The compilation process resembles that of Seq (Figure 5-1), but Codon additionally allows for extensions to the language syntax (i.e. adding new keywords), libraries (e.g. like Seq's `bio` module), and IR passes. These extensions are packaged as plugins that Codon can load and execute during compilation.

corporating new IR passes and syntax, which can be packaged as dynamic libraries and Codon source files. At compile time, the Codon compiler can load the plugin, registering the DSL's elements.

An illustration of how Codon's DSL plugins relate to the rest of the framework is given in Figure 8-1. In short, DSL plugins consist of three key elements:

- *Libraries:* Library modules relevant to the domain, such as the `bio` module in the case of Seq. This also includes types and functions that might be used by IR passes via bidirectional compilation.
- *Syntax:* Codon allows DSL plugins to register new keywords, and to define how these keywords are translated to (potentially new) IR nodes.
- *Optimizations:* Plugins can register domain-specific optimizations or analyses to be run on the IR.

These components are packaged into shared libraries that are dynamically loaded by

Codon, as well as a set of Codon source files for the library modules.

8.2.1 Extending the parser

Codon's parser is based on a *parsing expression grammar*, or PEG [50]. A set of core language rules are hard-coded into the grammar (i.e. to cover Python's base syntax). However, additional rules can be registered dynamically, with some constraints. Currently, the parser allows three classes of keywords to be registered:

- *Expression keywords* of the form `keyword` $\texttt{e}_1$ ... $\texttt{e}_n$ for expressions $\texttt{e}_1$ ... $\texttt{e}_N$.
- *Block keywords* of the form `keyword` $\texttt{e}_1$ ... $\texttt{e}_n$`: block`, where `block` is a block of code.
- *Binary keywords* of the form $\texttt{e}_1$ `keyword` $\texttt{e}_2$.

A DSL plugin must also specify how each new keyword is to be converted to IR nodes; this might include custom nodes, which we describe below.

8.2.2 Extending the IR

Many frameworks like MLIR [68] allow customization for all facets of the IR. While this allows for a great deal flexibility, it also comes at the cost of complexity when implementing new patterns, analyses, and transformations. Due to Seq's fully-featured base, we are able to restrict customization of the IR to a few types of nodes, ensuring compatibility with existing SIR infrastructure, all the while maintaining flexibility. In particular, SIR allows users to derive from "custom" types, flows, constants, and instructions, which interact with the rest of the framework through a declarative interface. For example, custom nodes derive from the appropriate custom base class (`CustomType`, `CustomFlow`, etc.) and expose a "builder" to construct the corresponding LLVM IR; we show a brief example of this API in Figure 8-2, which implements a 32-bit float (note that Seq's default `float` type is 64-bit). Notice that implementing custom types (and custom nodes in general) involves defining a `Builder` that specifies

```
class Builder : public TypeBuilder {
  llvm::Type *buildType(LLVMVisitor *v) {
    return v->getBuilder()->getFloatTy();
  }

  llvm::DIType *buildDebugType(LLVMVisitor *v) {
    auto *module = v->getModule();
    auto &layout = module->getDataLayout();
    auto &db = v->getDebugInfo();
    auto *t = buildType(v);
    return db.builder->createBasicType(
           "float_32",
           layout.getTypeAllocSizeInBits(t),
           llvm::dwarf::DW_ATE_float);
  }
};

class Float32 : public CustomType {
  unique_ptr<TypeBuilder> getBuilder() const {
    return make_unique<Builder>();
  }
};
```

Figure 8-2: 32-bit float `CustomType`

LLVM IR generation via virtual methods (e.g. `buildType` and `buildDebugType`); the custom type class itself defines a method `getBuilder` to obtain an instance of this builder. This standardization of nodes enables DSL constructs to work seamlessly with existing passes and analyses.

8.3 Examples

8.3.1 Seq

Since Seq is itself a DSL, it can be implemented as a plugin for Codon. In particular, all of Seq's core genomics-specific type (e.g. `seq` and `Kmer`) are implemented in Seq itself, so they can be packaged as a library along with the rest of the `bio` module. Seq's domain-specific optimizations can also be packaged in a shared library that uses the IR's API. Below we present new Codon-based DSLs for other domains with very different computational characteristics.

8.3.2 Sequre: a DSL for secure multi-party computation

Secure multi-party computation (MPC) [41] is a strategy for performing computations without actually revealing the underlying data. MPC operates by partitioning and distributing the data to multiple parties in such a way that no individual party is able to reconstruct the original values. These parties include data owners (which distribute shares of the data) and computing parties (which perform the computation of interest). Conceptually, the data owner *securely* shares their data, allowing the computing parties to perform secure computations like neural network training or statistical analyses. This computation is performed in a distributed manner, where each computing party only accesses its share of the original data, and occasionally communicates with its peers over a secure channel. Finally, after all shares have been computed, the parties broadcast their results back to the data owners, who then combine and reconstruct the shares into actual results. While this protocol is simple in theory, ensuring the security of a given computation or program requires careful

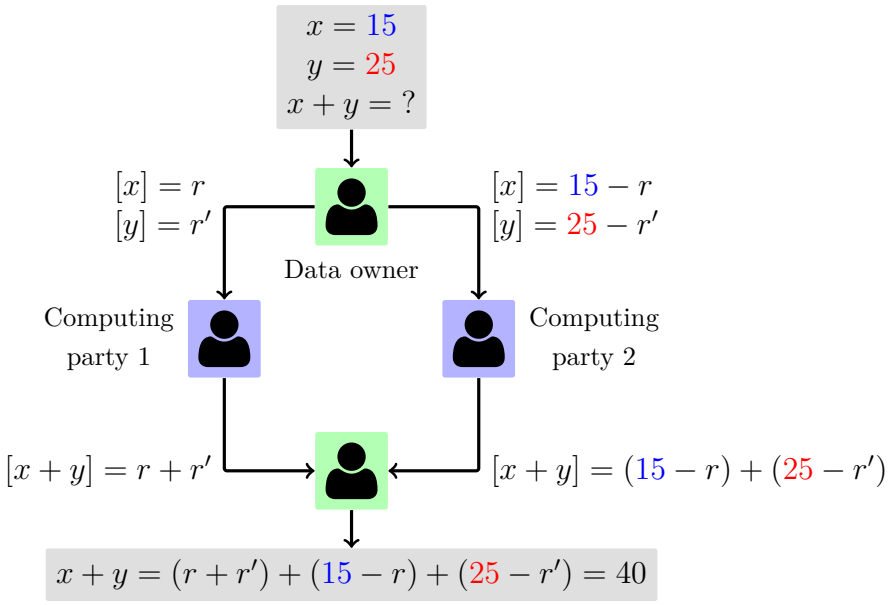


Figure 8-3: Example of secure multi-party computation for computing $x+y$ with two computing parties. The data owner (green) distributes the *shares* of both x and y to the computing parties (blue), which then compute the sum of their shares and send their results back to the data owner for reconstruction of the final sum.

code review, making simple, straightforward code preferable.

Data sharing and reconstruction relies on a randomization of data splits (or *shares*) to achieve security and hide the original data. For arithmetic operations like addition or subtraction, these shares can be obtained by a simple mechanism: in a scenario involving two computing parties and a secret value x, data can be split across the nodes by generating a random value r, assigning one node r and the other $x - r$ (written as $[r, x - r]$). Reconstruction of the original value can be performed by simply adding the shares: $r + (x - r) = x$. Since r is random, neither node has any information regarding the original value x. Basic arithmetic operations like additions and subtractions can be executed independently by each node. For example, if x and y are shared across two nodes as $[r, x-r]$ and $[r', y-r']$ respectively, we can simply add (or subtract) the shares at each node independently, obtaining $[r+r', (x-r)+(y-r')]$. Combining these yields the desired $x + y$. Both shares are still random, and as such do not invalidate the security guarantees of the protocol. An illustration of this procedure is given in Figure 8-3.

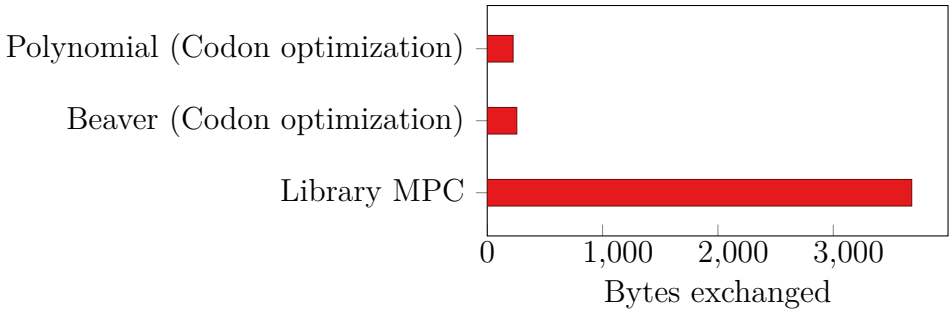


Figure 8-4: Total network utilization for standard MPC library ("Library MPC") and Sequre's two optimized solutions ("Beaver" and "Polynomial") in terms of exchanged bytes between computing parties.

Other arithmetic operations like multiplication are considerably more complicated, with the approach above yielding incorrect results. For example, simply multiplying shares independently results in an incorrectly reconstructed value of $(x - r)(y - r') + r \cdot r' \neq x \cdot y$. Numerous solutions to this problem have been discovered over the past decades [19, 33, 42]. The most common approach is known as a *Beaver multiplication protocol* [19], which rectifies the reconstructed value by adding $(x-r)r'$ to first share and $(y - r')r$ to the second. These values are called *Beaver partitions*, and calculating them incurs significant computational and communication overhead. Other arithmetic operations (such as division and exponentiation) also build on top of the Beaver protocol, and as such come with substantial overhead in both offline and online computing—with all nodes involved—as well as much more complex code.

Optimization efforts are usually focused on minimizing communication between computing parties, which is the largest bottleneck in the pipeline overall. Such optimizations include *Beaver partition caching* and *polynomial evaluation* [37]. The former attempts to reuse Beaver shares (that are expensive to generate) across different expressions, while the latter converts each arithmetic expression over shared data into canonical polynomial form, and is proven to induce minimal network overhead. However, implementing these optimizations manually requires significant code refactoring, and the optimizations themselves often obscure the underlying program, making the code much harder to review.

Sequre is a Codon-based DSL for secure multi-party computation that attempts to solve these challenges in a systematic way, improving performance by reducing network overhead, without sacrificing code simplicity. It is accompanied by a library that implements all state-of-the-art multi-party computing paradigms. Inspired by the above mentioned Beaver partition caching and polynomial evaluation optimizations, Sequre optimizes all multi-party arithmetic by leveraging Codon's IR to either transform all arithmetic operations on shared data to their cached multi-party equivalents, or by automatically converting such expressions to their polynomial forms, which are then evaluated with an optimized polynomial evaluation procedure (see Figure 8-5 for details). The first approach includes caching the Beaver partitions and powers for each variable, and propagating those caches as necessary, thus decreasing network overhead. These two approaches are mutually exclusive, and each can be superior in different scenarios. Namely, polynomial optimization is guaranteed to minimize network overhead between computing parties, but pays an extra cost in offline computation for polynomial expansion; its computational complexity is also exponential in terms of the polynomial's degree. Therefore it can become impractical even compared to a non-optimized solution if the polynomial's expansion is too large. We control this by statically calculating the size of the eventual polynomial expansion and, if it is too large, using the first optimization instead within the IR pass.

We implemented two mutually exclusive IR passes for optimizing arithmetic expressions in Sequre:

- *Beaver optimization:* Transforms arithmetic expressions into their secure multi-party counterparts that implement Beaver partitioning and powers caching at the same time

- *Polynomial optimization:* Reshapes arithmetic expressions into polynomial shape and scaffolds them for optimized polynomial evaluation

The Beaver optimization IR pass transforms all arithmetic operators to their multi-party counterparts optimized with an adapted Beaver caching paradigm. This pass,

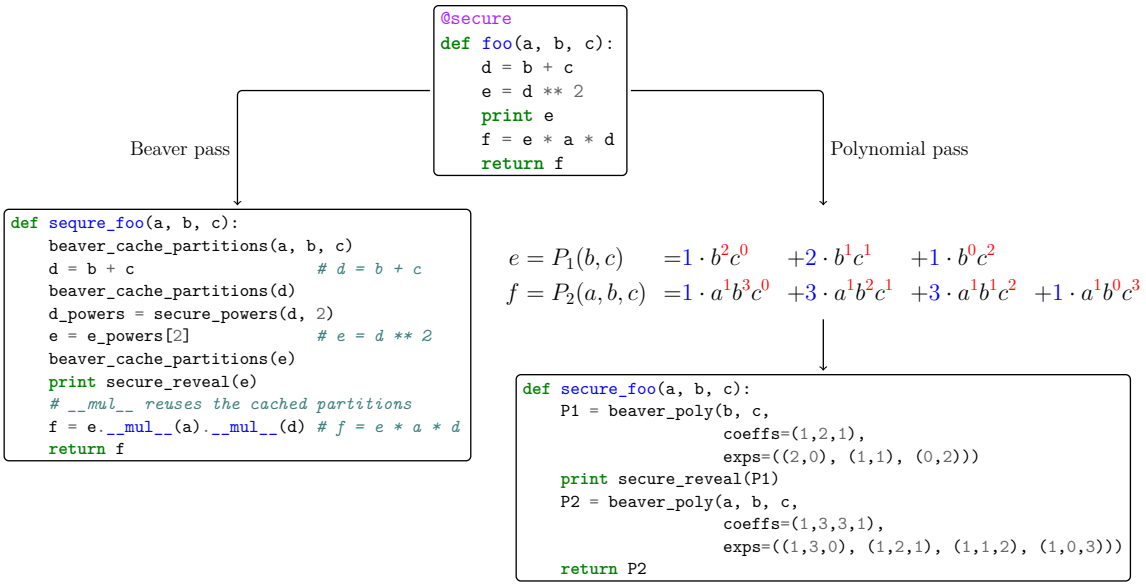


Figure 8-5: Sequre leverages Codon's IR to optimize arithmetic expressions via either the Beaver pass (left) or the Polynomial pass (right).

implemented in approximately 100 lines of C++ code, enables users to write secure multi-party arithmetic expressions in their simplest form without worrying about MPC optimizations. For instance, a hand-written implementation of the MPC-equivalent of the 3 assignments in `foo` from Figure 8-5, is 158 lines of code in Python (shown in simplified form in the bottom left). The polynomial optimization IR pass, on the other hand, expands all arithmetic expressions in a single block into their polynomial form and, similar to the previous pass, forwards the expanded polynomial coefficients and exponents to an optimized procedure for polynomial evaluation, reducing network overhead between the computing parties even further. Specifically, for the example shown in Figure 8-4, a state-of-the-art MPC library exchanges 3,680 bytes over the network, while Sequre decreases this to 256 bytes through the Beaver optimization approach, or to 224 bytes through the polynomial optimization approach.

8.3.3 CoLa: a DSL for block-based compression

Block-based data compression forms the core of many algorithms in use today, from the well-established JPEG standard for image compression, to the recently released

VVC/H.266 standard for video compression. Despite the importance of block-based compression, little attention has been given to providing language support, making implementation an arduous task. New compression algorithms are constantly being developed, with each iteration building on the last to improve compression performance and cover a wider variety of data types. Even though each version is fundamentally similar, developers often opt to re-implement from scratch in languages like C and C++, leading to complex codebases with hundreds of thousands of lines of code.

Much of the existing implementation complexity arises from a mismatch in the structures provided by existing languages and the structures needed for compression. The **Co**mpression **La**nguage, CoLa, is a Codon-based DSL that focuses on providing an intuitive, concise, expressive, and malleable interface for block-based compression implementations. CoLa focuses on three fundamental components of block-based compression that are not handled well in existing implementations: data representation, data traversals, and data partitioning.

Data representation is the core abstraction in CoLa, providing native multi-dimensional data types called `Block`s and `View`s for handling the data in block-based compression. Despite the wide variety of use cases handled in CoLa, the implementation in Codon is simple, making heavy use of existing Codon features such as generics and method overloading to provide a straightforward implementation.

The data traversal abstractions deal with how the individual elements in a `Block` and/or `View` are traversed, often using complex patterns not easily described with loop nests. The bottom of Figure 8-6 shows a 3$^{\text{rd}}$-order Hilbert space-filling curve, which is one type of traversal pattern used in medical image compression [7]. CoLa defines a traversal as a series of step and rotation operations which dictate how to move from one coordinate within a grid to the next. The top left of the figure gives the CoLa code for an arbitrary order Hilbert curve. Codon makes this notation possible thanks to its support for annotations and its flexible front-end. The `@traversal` annotation is recognized in the compiler, where the compiler automatically inserts ad-

```
@traversal
def hilbert(order: int, i: int=0,
            which: bool=True):
  tparams 2
  if i < order:
    rrot 90 if which else 270
    link hilbert(order, i+1, not which)
    rstep 1
    rrot 270 if which else 90
    link hilbert(order, i+1, which)
    rstep 1
    link hilbert(order, i+1, which)
    rrot 270 if which else 90
    rstep 1
    link hilbert(order, i+1, not which)
    rrot 90 if which else 270
```

```
void hilbert(int n, int *a) {
  int i1,j1,i;
  int b[(int)pow(4,n)];
  int tmp;
  if (n==1) {
    tmp = a[2];
    a[2] = a[3];
    a[3] = tmp;
  } else {
    for (i1=0; i1<(int)pow(2,n-1); i1++) {
      for (j1=0; j1<(int)pow(2,n-1); j1++) {
        b[j1*(int)(pow(2,n-1))+i1] =
          a[i1*(int)(pow(2,n))+j1];
        b[i1*(int)(pow(2,n-1))+j1+(int)pow(4,n-1)] =
          a[i1*(int)(pow(2,n))+j1+(int)pow(2,n-1)];
        b[((int)pow(2,n-1)-1-j1)*(int)(pow(2,n-1)) +
          ((int)pow(2,n-1)-1-i1) + 3*(int)pow(4,n-1)] =
          a[i1*(int)(pow(2,n)) + j1 +(int)pow(2,2*n-1)];
        b[i1*(int)(pow(2,n-1))+j1+2*(int)pow(4,n-1)] =
          a[i1*(int)(pow(2,n))+j1+(int)pow(2,2*n-1)+
            (int)pow(2,n-1)];
      }
    }
    for (i=0; i<(int)pow(4,n); i++) a[i] = b[i];
    hilbert(n-1, &a[0]);
    hilbert(n-1, &a[(int)pow(4,n-1)]);
    hilbert(n-1, &a[2*(int)pow(4,n-1)]);
    hilbert(n-1, &a[3*(int)pow(4,n-1)]);
  }
}
```

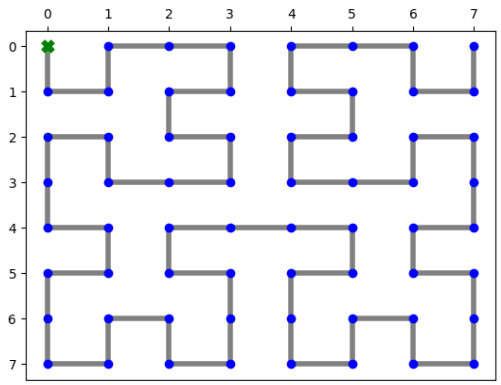
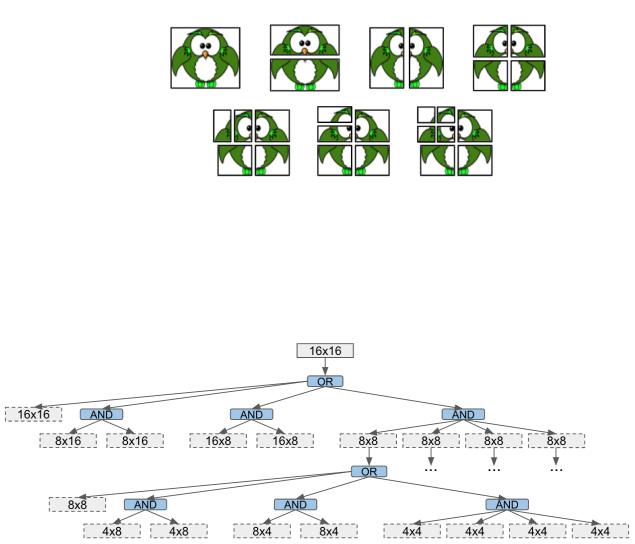


Figure 8-6: CoLa implementation (top left) and C implementation [78] (top right) for a Hilbert space-filling curve (bottom).

ditional boilerplate information into the traversal definition. The keywords `tparams`, `rrot`, `rstep`, and `link` are some of the custom CoLa keywords added into Codon which control the step and rotation operations defining the traversal. CoLa's traversal syntax provides a much simpler and intuitive way to define patterns compared to existing imperative implementations, such as the recursive loop-based C implementation in the top right of Figure 8-6.

The final CoLa abstraction, data partitioning, deals with defining how to split a `Block` or `View` into smaller non-overlapping blocks. In block-based compression, there can be thousands of different ways to partition a single block. Existing implementations implicitly create these partitions on-the-fly, where the type of partition is completely obscured by the surrounding code. CoLa provides a unique representation for partitions based on *And-Or trees* from artificial intelligence [80], where *And* nodes represent a set of non-overlapping blocks derived from a parent block, and *Or* nodes define different options for partitioning a parent. The top left of Figure 8-7 illustrates a partition from AVC/H.264 video compression, showing 7 of the 259 possible ways to partition a block. The lower left of the Figure gives the And-Or tree representation of this. Similar to traversals, CoLa utilizes Codon's annotations and also adds custom keywords to provide a simple syntax for defining partitions, with the CoLa code for defining all 259 partitions shown on the right of Figure 8-7. `@ptree` signals to the compiler that this is a partition definition, and `pparams`, `pt_or`, `pt_and`, and `pt_leaf` are custom keywords for constructing the And-Or tree.

Thanks to the wide variety of programming features already available in Codon, as well as its flexibility for introducing custom syntax, CoLa is able to provide much needed language support for implementing the complex features of block-based compression that complicate existing implementations.




```python
@ptree
def macroblock_partition(root):
  # set the root of the tree
  pparams root
  M = root.dims(0)
  pt_or:
    pt_leaf root
    pt_and:
      pt_leaf root[:,:M//2]
      pt_leaf root[:,M//2:]
    pt_and:
      pt_leaf root[:M//2,:]
      pt_leaf root[M//2:,:]
    pt_and:
      q0 = root[:M//2,:M//2]
      q1 = root[:M//2,M//2:]
      q2 = root[M//2:,:M//2]
      q3 = root[M//2:,M//2:]
      if M == 16:
        link macroblock_partition(q0)
        link macroblock_partition(q1)
        link macroblock_partition(q2)
        link macroblock_partition(q3)
      else:
        pt_leaf q0,q1,q2,q3
```

Figure 8-7: The top left shows 7 of the 259 possible partitions for a 16×16 block are shown in the top left. The bottom left shows an And-Or tree representing all 259 partitions (where "..." denotes the same sub-tree as on the left). The code on the right shows the equivalent CoLa implementation for the `PTree`.

8.4 Conclusion

Although Seq has been designed from the ground up with genomics and bioinformatics in mind, it is still a powerful tool even for general-purpose computing. Codon extends that notion by factoring out Seq's domain-specific components, and augmenting the compiler with a plugin architecture. Seq's intermediate representation in turn enables the system to be extended with new optimizations, analyses, data types, and even control-flow structures for new domains. Thereby, Codon aims to have the same effect that Seq has in genomics, albeit in a range of new areas.

Chapter 9

Applications and Results

9.1 End-to-End Applications

A typical genomics analysis pipeline currently consists of four fundamental types of stages: (i) data pre- and post-processing, (ii) reference sequence processing, (iii) read sequence processing, and (iv) downstream analysis. Each of these stages employs different classes of algorithms and has distinct performance challenges (e.g. processing of reads often involves memory-bound operations whereas downstream analyses are frequently numerically-intensive). Thus, unique optimization techniques must be used at each stage in order to achieve ideal scalability and performance.

To demonstrate the utility of Seq, we re-implemented eight standard, real-world applications to evaluate the performance of the compiler's domain-specific optimizations and the resulting code complexity across these stages, as compared to existing highly-optimized and widely-used C, C++, Java or Python implementations—and, in some instances, their equivalents using Rust, Julia, and C++ genomics libraries. Here we describe each of these applications in greater detail, as well as what is gained by porting them to Seq. An overview of these applications in the context of the genomics pipeline is given in Figure 9-1, which also shows the speedups attained by Seq.

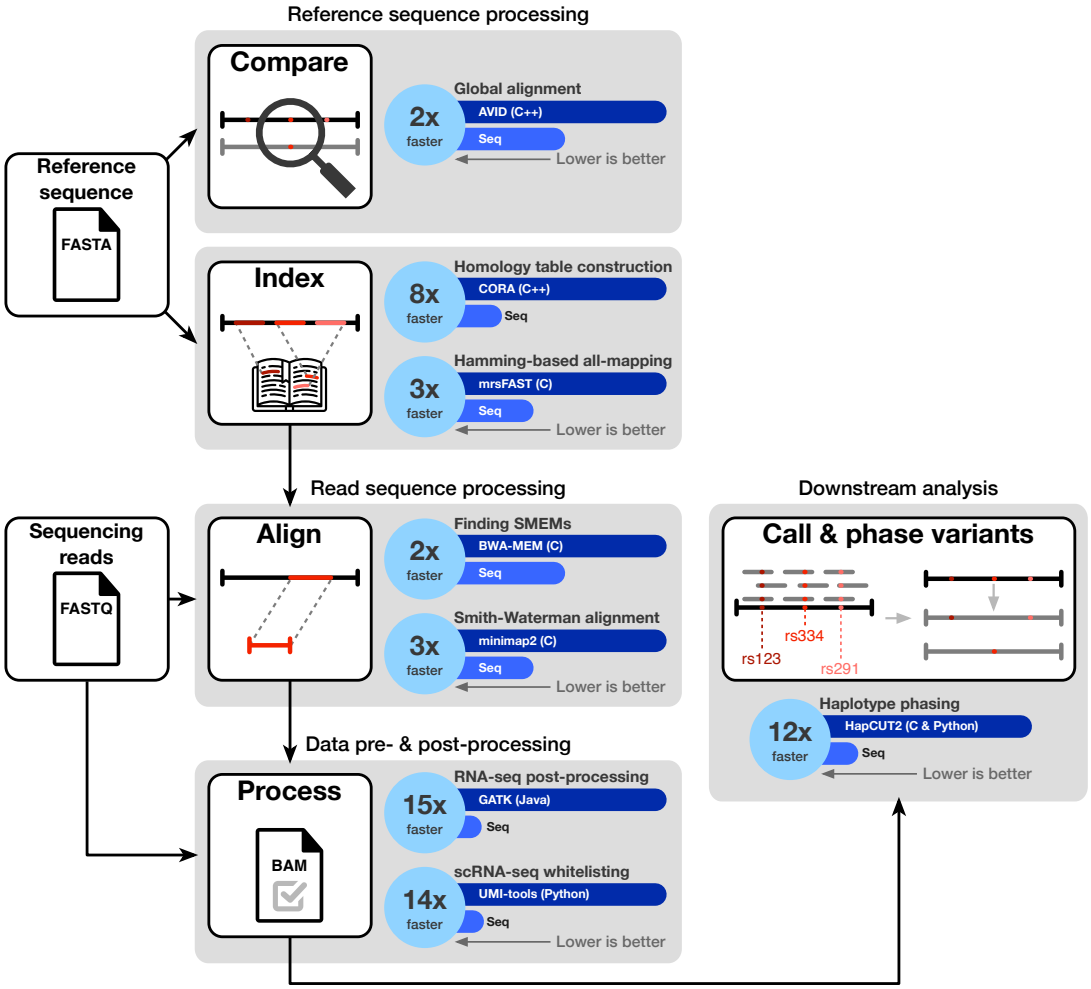


Figure 9-1: Seq performance improvements demonstrated on eight common, state-of-the-art applications. Comparisons based on a typical genomics pipeline using the Seq language and compiler to implement applications: global alignment (from AVID) under "Compare" for comparative genomics; homology table construction (from CORA) and Hamming distance-based all mapping (from mrsFAST, which also performs alignment) under "Index" for reference sequence indexing; finding super-maximal exact matches, or SMEMs (from BWA-MEM) and Smith-Waterman alignment (from minimap2) under "Align" for read sequence alignment; RNA-seq read post-processing (from GATK `SplitNCigar`) and single-cell RNA-seq barcode whitelisting (from UMI-tools) under "Process" for data pre- and post-processing; and haplotype phasing under "Call & phase variants" for numerically-intensive downstream analysis. The accompanying bar charts show runtimes of the original tools compared to the Seq versions.

Before we begin, let us provide a few notes about the re-implementations described below. First, all the tools we compared against are well-tested and highly optimized (often by a large team of developers), and provide several additional options aside from the functionality being compared against in this work. Our goal here is to show that the same—or even *much* better—results can be achieved in a fraction of the time and development effort with a high-level programming language given the proper compiler support.

To ensure fairness, all of the reported results, unless otherwise specified, were generated in single-threaded mode, and unless explicitly specified, the outputs produced by our re-implementations exactly match those of the tools being compared against.

Finally, we would like to note that our memory measurement metric—the maximum resident memory used by a process—is far from perfect. This metric is heavily impacted by system-specific memory management; for example, macOS automatically compresses the working set of high-memory processes, and thus reports lower memory usages (a side-effect of this strategy is occasional slow-down of such processes). Furthermore, this metric struggles with garbage collection-based runtimes such as Seq's, Java's, Python's and Julia's, as it often reports memory usages higher than the actual value.

In the results below, runtimes are reported in an "hh:mm:ss" format (or "mm:ss" or simply "ss" for shorter runtimes), and memory usages are given in gigabytes. Experiments for which memory usages are not reported used less than 1 GB of RAM. "Seq imprv." refers to Seq's runtime improvement.

Benchmarking details All experiments in this section were run on the following systems:

1. dual-socket system with Intel Xeon Gold 5218 CPUs (2.30 GHz) with 16 cores each (totalling 32 cores and 64 hyper-threads) and 768GB DDR4-2933 RAM and Linux OS.

2. desktop iMac with Intel Core i9-9900K CPU (3.60 GHz) with 8 cores (totalling 16 hyper-threads) and 64GB DDR4-2667 RAM and macOS Big Sur.

The Seq version is 0.10.1. `https://github.com/seq-lang/seq-benchmarks` contains detailed instructions for reproducing the results in this section.

9.1.1 Reference sequence processing

Genome homology table construction (CORA) CORA [128] is an all-mapping tool for next-generation sequencing (NGS) reads. In other words, given a read (or a read pair), CORA reports all alignments (up to some edit or Hamming distance) of that read in a reference sequence. To achieve this goal, the primary data structure used by CORA is the *homology table*, which stores groups of "homologous" regions in the reference, and is used to convert a single alignment as reported by an off-the-shelf mapper like BWA, into a list of all mappings satisfying the distance criteria. Two versions of the homology table are built: *exact* and *inexact*. The exact homology table stores in a compressed form pairs of equal k-mers (with $33 \leq k \leq 64$) from the reference, along with their loci (where "compression" is done by merging two or more consecutive homologous pairs into a single group). Similarly, the inexact homology table stores pairs of unequal k-mers whose Hamming distance is less than some threshold. CORA's definition of homology also allows for cases where one k-mer is homologous to the reverse complement of another k-mer.

The process of constructing the homology table involves many of the operations that Seq optimizes, including k-merization, reverse complementation, k-mer matching and Hamming distance calculations. Indeed, we implemented CORA's homology table construction in Seq, and compared both performance and lines of code with the original, highly-optimized C++ implementation. We used a k-mer length of 64 and a maximum Hamming distance threshold of 1 for the inexact homology table. Exact homology table construction was single-threaded, whereas the inexact table's construction was allowed to use all 48 threads of our machine.

Genome homology table construction (CORA)

Linux	Time	Memory	Seq imprv.	Time	Memory	Seq imprv.
CORA (C++)	1:32:09	206.7	7.9×	11:42:44	329.6	3.7×
Seq	**11:44**	**72.5**	—	**3:07:47**	**50.8**	—
	(a) Exact matching			*(b) Inexact matching*		

Table 9.1: Performance improvements in homology table construction from using the Seq language and compiler. Due to the memory requirements, we only ran this experiment on a large-memory Linux machine. The Seq version achieves nearly an 8× speed improvement while using 3–6× less memory than the original version.

Table 9.1 shows the results of the comparison. The Seq versions were substantially smaller in terms of lines of code (over 2,500 in the original C++ version versus less than 300 in Seq—nearly 10× fewer lines) and ran substantially faster (4–8×) while using 3–4× less memory, which has been somewhat of a barrier to CORA's use (both CORA and the Seq re-implementation were run with 24 threads for inexact construction). Lastly, we found the outputs of the original and Seq implementations to be very slightly different; in particular, the original version found 4,186,875 exact homologies in hg19 with a k-mer length of 64, whereas the Seq implementation found 4,372,977 (roughly 4% more) for the same k-mer size; all of the additional homologies were found to be correct by a manual inspection. The additional homologies have now been made publicly available at `https://github.com/seq-lang/seq-benchmarks`. Therefore, we attribute this discrepancy to a bug in the original version. Note that while such bugs are hard to find in large and tightly optimized C/C++ codebases, they often become self-evident in high-level codebases that focus only on the core algorithm, such as those produced with Seq.

Global sequence alignment (AVID) AVID [29] is one of the fastest tools for global alignment of large nucleotide and amino acid sequences (see also Batzoglou et al. [18]). Although the canonical global alignment algorithm (Needleman-Wunsch) is a slight variant of the local alignment algorithm (Smith-Waterman), only a few tools such as FASTA [96] and AVID (released in 1989 and 2002, respectively) are able to quickly produce global alignments between two sequences. Unfortunately, the source

code of AVID is not available, and pre-built binaries are only available for a limited number of platforms.

We re-implemented AVID's algorithm as described in the original publication [29]. The core of AVID's method consists of finding "Maximal Exact Matches" (MEMs)—identical sub-sequences between the two sequences of interest that cannot be further extended—and using them to guiding the global alignment process. MEMs are commonly found by recursively traversing a generalized suffix tree made from the subject sequences. In our implementation, we used suffix arrays (SA) instead of suffix trees (as Seq supports efficient SA operations out of the box), and a bottom-up traversal algorithm over SAs to find all MEMs [4].

Overall, we were able to prototype the whole pipeline as described in the paper in 170 lines of high-level Seq code. We ran AVID and Seq-AVID on the large set of similar (similarity $\geq$90%) and less-similar sequences ($\geq$70%) from the human genome segmental duplications database [16, 92], and found that the Seq implementation provided a 2$\times$ speedup over the original binary while maintaining a similar accuracy (note that our re-implementation is not an exact reproduction of the original software as the exact algorithm used by the currently available AVID binary differs slightly from the published description; nevertheless, our implementation produces alignments that are highly concordant with the alignments produced by the AVID binary). The results of this comparison are shown in Table 9.2.

9.1.2 Read sequence processing

Finding SMEMs (BWA-MEM) Super-Maximal Exact Matches, or SMEMs, are a subset of MEMs such that no two SMEMs overlap within the read sequence. SMEMs are an integral component of many important genomics algorithms, ranging from sequence alignment to assembly [72, 73, 122]. Because the SMEM detection algorithm primarily involves querying an FMD-index data structure (unlike the MEM detection algorithm, which relies on suffix tree traversal), it is inherently memory-bound, and can be greatly accelerated by several of Seq's compiler optimizations.

Global sequence alignment (AVID)

Linux	Time	Memory	Seq imprv.	Time	Memory	Seq imprv.
AVID (C/C++)	35:05	**1.0**	2.0×	34:24	**2.0**	2.0×
Seq	**17:44**	2.2	—	**16:52**	5.3	—
	(a) 90% similarity			*(b) 75% similarity*		

Table 9.2: Performance improvements in global sequence alignment from using the Seq language and compiler. As we only had access to the Linux binary distribution of AVID, we had to restrict this benchmark to the Linux machine. The Seq version achieves nearly a 2× speed improvement. Note that the results are not identical, as the original source code is not available. However, we found out that $\geq 90\%$ of all generated alignment scores are nearly identical (within 10% difference) between the tools.

To this end, we implemented BWA-MEM's SMEM-finding algorithm in Seq, with roughly 135 lines of code (compared to the over 500 lines of C code comprising the original). The standard algorithm involves repeatedly querying an FMD-index to extend matches between a read and the reference, which causes expensive memory stalls—indeed, this is the main bottleneck in the algorithm. As querying large genomic indices is itself a prevalent operation in genomics algorithms, the Seq compiler automatically performs several optimizations to greatly speed up such queries. Rather than issuing memory requests serially and forcing the entire program to stall, Seq makes use of coroutines and software prefetching to *overlap* the memory-bound stall from one query with other useful work, greatly reducing the effective stall time. This process is shown in Figure 6-4.

While implementing these transformations by hand in existing codebases would require substantial effort (including implementing coroutines, scheduling and managing coroutine state, dispatching queries and returning results, debugging, etc.), in Seq it only requires a single additional `@prefetch` annotation—everything else is handled automatically by the compiler. Applying this optimization to the SMEM algorithm resulted in a roughly 2× speedup over BWA-MEM (Table 9.3).

Software prefetching is also used in BWA-MEM2, the next iteration of BWA-MEM, which adds numerous manual, low-level performance optimizations [122]. However,

BWA-MEM2's implementation does not perform a coroutine-based software context switch like Seq compiler does; consequently, as the authors describe, it cannot fully remedy memory-bound stalls. By contrast, Seq's use of coroutines to overlap stalls with useful work can often ensure that a particular address is cache-resident by the time it needs to be accessed (i.e. when a suspended coroutine is resumed, as described in detail in Methods). We also evaluated software prefetching *without* the use of coroutines—much like what BWA-MEM2 does—but observed no appreciable performance improvements over the coroutine-based prefetching.

In order to compare the performance of Seq with other high-performance genomics libraries, we also implemented this experiment in Rust using the Rust-Bio library [65] and in C++20 using the SeqAn3 library [45]. In both cases, the Seq version takes roughly 6–9× less time to process the reads than the alternate implementations, regardless of whether Seq's prefetch optimization had been enabled. Finally, we would like to point out that the Seq version is also cleaner and more adaptable: Rust-Bio had to be manually patched[1] to support exact SMEM extraction, while the SeqAn version had lengthier code, was more verbose, and took 6× longer to compile.

Hamming-distance based all-mapping (mrsFAST) mrsFAST [54] is a perfect-sensitivity all-mapping tool based on Hamming distance. mrsFAST employs a hash table to index k-mers from a reference genome, and uses that table to find all mappings of a given read that are below a user-defined Hamming distance threshold e. The algorithm is centered around the pigeonhole principle: each read is partitioned into $e + 1$ non-overlapping seeds, which guarantees that at least one seed will map to all positions in the reference at which the Hamming distance is at most e. mrsFAST-Ultra, the latest version of the tool, is implemented in C, and involves many of the operations discussed here, including k-merization, reverse complementation, k-mer hashing, indexing and of course Hamming distance calculations.

We implemented a subset of mrsFAST-Ultra for single-end mapping in Seq to further

[1] `https://github.com/rust-bio/rust-bio/pull/392`

Finding SMEMs (BWA-MEM)

Linux	Time	Memory	Seq imprv.
BWA (C)	1:37	**0.4***	1.7×
Rust-Bio (Rust)	6:23	81.2	6.7×
SeqAn (C++)	5:16	2.5	5.5×
Seq	1:01	7.2	1.1×
Seq (prefetch)	**57**	7.2	—

macOS	Time	Memory	Seq imprv.
BWA (C)	1:19	**0.4***	2.3×
Rust-Bio (Rust)	4:17	61.1	7.6×
SeqAn (C++)	4:53	2.6	8.6×
Seq	50	7.3	1.5×
Seq (prefetch)	**34**	7.3	—

Table 9.3: Performance improvements in SMEM finding from using the Seq language and compiler on Linux and macOS systems. The Seq version achieves nearly a 2× speed improvement over BWA, and roughly a 6–9× improvement over high-performance genomics libraries Rust-Bio and SeqAn. The reported timings represent the time needed for each tool to find SMEMs in a set of FASTQ reads; index building and loading times are not included, as they vary across the tools. Note that Seq, Rust and SeqAn FM-index implementations are not compressed, and as such use more RAM than BWA's compressed implementation (marked with *). Also note that prefetch improvements depend on the CPU cache size: large-cache machines (such as our Linux machine) benefit less than machines with smaller CPU caches (macOS).

test the effectiveness of the compiler's optimizations, the results of which are shown in Table 9.4. With a distance threshold $e = 2$ on chromosome 1, the Seq implementation was ≈30% faster than the original hand-optimized C implementation, with a third of the code size. We also implemented a version using an FM-index instead of a hash table—a modification that requires changing only a few lines of code in Seq—and tested it with exact matching ($e = 0$) on the entire genome, resulting in a 2.5× improvement, which is particularly notable given that mrsFAST-Ultra's is a cache-oblivious algorithm. Ultimately, the original version is comprised of well over 1000 lines of code and takes over an hour to perform exact matching, whereas the Seq version is comprised of 100 lines of code and completes in about 15 minutes.

Hamming-distance based all-mapping (mrsFAST)

Linux	Time	Memory	Seq imprv.	Time	Memory	Seq imprv.
mrsFAST (C)	10:39	**13.1**	1.3×	27:03	34.5	2.7×
Seq	**8:30**	17.8	—	17:25	**25.1**	1.7×
Seq (prefetch)		N/A		**10:11**	25.2	—

macOS	Time	Memory	Seq imprv.	Time	Memory	Seq imprv.
mrsFAST (C)	8:47	**13.2**	1.1×	17:41	34.1	1.3×
Seq	**8:08**	19.0	—	20:33	**25.1**	1.6×
Seq (prefetch)		N/A		**13:17**	25.2	—

(a) Inexact matching ($e = 2$, chr1) *(b) Exact matching (hg19)*

Table 9.4: Performance improvements in Hamming distance-based read alignment from using the Seq language and compiler on Linux and macOS systems. The Seq version achieves up to a 2.5× speed improvement over mrsFAST. The reported timings represent the time needed for each tool to find all-mappings for a set of FASTQ reads; index building and loading times are not included. Note that Seq uses mrsFAST's k-mer index for inexact matching, and an FM-index for exact matching, which explains the somewhat varying RAM usages across the tools in different experiments.

Long-read mapping (minimap2) As third-generation sequencing becomes more widespread, high-performance methods and tools for processing long-reads become increasingly important [112]. minimap2 [74] is the current state of the art in terms of long-read alignment, both in terms of performance and accuracy. The algorithm used by minimap2 follows similar steps as AVID, and involves chaining to find overlaps between reads and the genome, followed by dynamic programming alignment between the seeds of a chain to obtain the final alignments.

Smith-Waterman sequence alignment is an essential kernel in many genomics applications, and is consequently a heavily researched area [48, 104, 74]. Most hand-optimized implementations use instruction-level SIMD parallelism to accelerate the alignment of a single pair of sequences—an approach referred to as intra-sequence alignment. However, yet another approach is to use SIMD to accelerate aligning *multiple* pairs of sequences, referred to as inter-sequence alignment [104]. While inter-sequence alignment can be substantially faster than intra-sequence alignment, it is typically cumbersome to implement with general-purpose programming languages due to the need to batch sequences, manage state, dispatch alignment results and

so on. In Seq, however, the compiler performs several pipeline transformations to convert standard alignment to inter-sequence alignment with just a single-line code annotation.

We implemented minimap2's Smith-Waterman alignment step in Seq to test the compiler's inter-sequence alignment optimization relative to minimap2's SIMD-optimized intra-sequence alignment kernel (in fact, Seq's default alignment kernel is the same one used by minimap2; minimap2 does not support inter-sequence alignment, however). Results of this comparison are shown in Table 9.5, where Seq's inter-sequence alignment optimization is up to 2.5× faster than the intra-sequence implementation. Note that the traceback step (required to produce CIGAR strings) is not vectorized in minimap2 nor Seq, meaning timings including CIGAR generation are expected to be comparatively strictly slower than those without CIGAR generation. Additionally, Seq automatically demotes sequences that do not benefit from inter-sequence optimization[2] to intra-sequence alignment (i.e., the same method is used to align such pairs in both the Seq implementation and in minimap2), so consequently timings including "all" sequences will also be comparatively slower. Nevertheless, even when including CIGAR generation and all sequence pairs, Seq performs up to 45% better than intra-sequence alignment.

We also implemented a simplified version of this experiment in Rust using Rust-Bio, SeqAn3, and Julia using the BioJulia library [126], in order to compare our implementation with highly-optimized genomics libraries. For a fair comparison, we just compared the speed of standard global alignment without the additional parameters that Seq offers (e.g. bandwidth, Z-drop and a couple other parameters were disabled as other libraries do not support them). Seq is roughly 7–34× faster than the alternatives, even without the inter-sequence alignment optimization. In more concrete terms, Seq can find the full alignments for 50,000 sequence pairs in a few seconds, as compared to couple minutes needed by the other tools. As such, the Seq version is

[2]We found that sequences >512 bases did not benefit from inter-sequence alignment; this limit is adjustable.

substantially more scalable on real-world datasets.

Long-read mapping (minimap2; ≈1 million sequences)

Linux	CIGAR	Score-only	CIGAR	Score-only	Seq imprv.
minimap2 / KSW2 (Seq/C)	1:28	1:14	55	46	1.3–2.6×
Seq (SSE4 inter-align)	1:23	1:02	50	35	1.2–1.9×
Seq (AVX2 inter-align)	1:07	51	30	23	1.0–1.3×
Seq (AVX-512F inter-align)	**1:07**	**50**	**27**	**18**	—

macOS	CIGAR	Score-only	CIGAR	Score-only	Seq imprv.
minimap2 / KSW2 (Seq/C)	1:06	56	41	34	1.5–2.3×
Seq (SSE4 inter-align)	59	47	34	26	1.3–1.7×
Seq (AVX2 inter-align)	**45**	**36**	**20**	**15**	—
	All sequences		*Small sequences (≤ 512)*		

Long-read mapping (minimap2; 50,000 sequences)

Linux	CIGAR	Score-only	Seq imprv.
minimap2 / KSW2 (Seq/C)	16	12	1.1×
Seq (AVX-512F inter-align)	**15**	**11**	—
Rust-Bio (Rust)	6:22	3:16	19–25×
SeqAn (C++)	3:13	2:17	12–13×
BioJulia (Julia)	1:52	1:52	7–10×

macOS	CIGAR	Score-only	Seq imprv.
minimap2 / KSW2 (Seq/C)	12	10	1.1×
Seq (AVX2 inter-align)	**11**	**9**	—
Rust-Bio (Rust)	6:10	3:16	21–34×
SeqAn (C++)	2:35	1:52	12–14×
BioJulia (Julia)	1:24	1:24	7– 9×

Table 9.5: Performance improvements in long-read mapping and sequence alignment from using the Seq language and compiler on Linux and macOS systems. Seq versions achieves up to a 2.5× speed improvement over standard SIMD-optimized alignment algorithms, and up to a 34× improvement over their non-SIMD counterparts. The reported timings represent the average time for 3 separate runs to account for SIMD-based time variation. "CIGAR" refers to CIGAR string generation (i.e. the backtrace step), and the "≤ 512" experiments were run on sequences shorter than that limit. Seq's inter-sequence optimization ("inter-align") was applied on several instruction sets (SSE4, AVX2 and AVX-512F). Our macOS machine did not support the AVX-512F instruction set.

One distinct advantage of the Seq language over general-purpose languages is the ability to incorporate complex pipeline optimizations—like those for inter-sequence alignment and prefetching—with minimal code changes. For example, adding inter-sequence alignment to the current C implementation of minimap2 would require large-

scale code refactoring to batch sequences (and other state information) before performing the alignment, and the complete rewrite of a complex SIMD-based alignment kernel, whereas in Seq all of this is handled implicitly by the compiler using coroutines, and usually requires only a single-line code annotation, as shown in Figure 6-8.

9.1.3 Data pre- and post-processing

RNA-seq data clean-up (GATK `SplitNCigar`) The most time-consuming steps in a NGS (next-generation sequencing) processing pipeline often consist of mundane data transformations performed on large datasets prior to, during and after the alignment or downstream analysis steps [93]. These tasks typically employ string operations (e.g. duplicate marking, sample tagging), score recalculations (e.g. base quality score re-calibration), and in the case of RNA-seq samples, splitting reads that span large intronic regions. Such tasks are commonly performed by GATK and Picard tools [30, 87]. However, these tools leave a lot to be desired in terms of performance, mainly due to the high-level language they are written in (Java). While data HLpost-processing tasks are often conceptually simple, the sheer volume of data (often on the order of terabytes) heavily penalizes even minor performance shortcomings that high-level languages often bear (e.g. an unnecessary memory copy can easily add several additional hours to the total runtime on real-world datasets). For instance, a relatively simple tool for splitting intron-spanning reads—`SplitNCigar`—from the GATK suite takes more than 10 hours to complete on a medium-sized 20GB BAM file.

We show that such post-processing can be done faster and easier by re-implementing GATK's `SplitNCigar` tool in Seq. By relying on automatic, conservative memory management and thus avoiding unnecessary memory copy operations, we were able to obtain 14× faster runtimes while using up to 10× less memory than the original GATK implementation (note that both Java and Seq rely on a garbage collector for managing memory). In concrete terms, this means that a 10-hour post-processing step can be completed in a less than an hour with Seq. Results of this comparison

RNA-seq data clean-up (GATK `SplitNCigar`)

Linux	Time	Memory	Seq imprv.	Time	Memory	Seq imprv.	Time	Memory	Seq imprv.
GATK (Java)	2:30:55	11.0	14.5×	7:29:52	11.8	9.8×	10:09:08	11.6	10.4×
Seq	**10:23**	**0.7**	—	**45:54**	**4.0**	—	**58:36**	**2.0**	—

macOS	Time	Memory	Seq imprv.	Time	Memory	Seq imprv.	Time	Memory	Seq imprv.
GATK (Java)	1:42:06	8.2	13.4×	5:05:12	10.0	10.4×	8:28:29	11.7	12.6×
Seq	**7:38**	**0.6**	—	**29:16**	**4.1**	—	**40:19**	**1.9**	—
	(a) 1.8 GB BAM			*(b) 13 GB BAM*			*(c) 20 GB BAM*		

Table 9.6: Performance improvements in RNA-seq data post-processing from using the Seq language and compiler on Linux and macOS systems. The Seq version achieves up to a 14× speed improvement over GATK `SplitNCigar` with significantly lower memory usage. Both tools were run on single-ended BAM files with no optional tags.

Single-cell data pre-processing (UMI-tools)

Linux	Time	Memory	Seq imprv.
UMI-tools (Python)	1:01	0.1	10.2×
Seq	**6**	**0.0**	—

macOS	Time	Memory	Seq imprv.
UMI-tools (Python)	56	0.1	14.0×
Seq	**4**	**0.0**	—

Table 9.7: Performance improvements in single-cell barcode whitelisting from using the Seq language and compiler on Linux and macOS systems. The Seq version achieves a ≥10× speed improvement over UMI-tools.

are shown in Table 9.6.

Single-cell data pre-processing (UMI-tools) Single-cell sequencing technologies typically employ short sequence barcodes to differentiate cells within the sequenced sample. However, a sequencing dataset often contains more barcodes than the sequenced cells due to sequencing errors and biases. Thus, the first step in any single-cell RNA-seq pipeline consists of whitelisting *true* barcodes that identify cells, and discarding others based on abundance levels. This task is commonly performed with Python-based tools such as UMI-tools [116].

We have ported this pre-processing step to Seq in less than 90 lines of code, achieving more than a 10× speedup over the original version. Results of this comparison are shown in Table 9.7.

9.1.4 Downstream analysis

Haplotype phasing (HapTree-X) Haplotype phasing is the process of separating variants into a set of maternal and paternal alleles, and can provide valuable biological insights beyond what can be gained from raw genotypes alone [6]. HapTree-X is a phasing algorithm implemented in Seq on top of HapTree [23], which utilizes Bayesian optimization to infer optimal haplotypes from aligned genomic data. In contrast to other case studies that focus mostly on discrete optimization and sequence operations, the HapTree-X phasing algorithm is a probabilistic optimization framework that spends most of its runtime doing numerical computations and graph traversals. Furthermore, HapTree-X is a complete genomics analysis pipeline that needs to preprocess and analyze large BAM files prior to the haplotype phasing step. As such, it is an excellent test of Seq's versatility and applicability to applications that require both efficient sequence processing *and* high-performance numerical algorithms.

Performance of Seq/HapTree-X on three diverse datasets spanning multiple sequencing technologies, relative to another two state-of-the-art phasing tools, is shown in Table 9.8. HapTree-X was faster than phASER (Python), and had comparable performance to that of HapCUT2 (C), on smaller RNA-seq and medium-sized exome datasets. However, we found that the performance gains increased with the scale of the data: on a large 120GB WGS BAM, HapTree-X was 3× faster than HapCUT2, while on a high-coverage 10X Genomics dataset (≈200GB BAM file), HapTree-X achieved up to a 12× speed up over HapCUT2 (C++, Python). Furthermore, parallelizing the HapTree-X pipeline and running it on four threads (by simply converting a few nested function calls into a Seq pipeline) further decreased the total runtime over 2×, totalling in a nearly 25× speed increase. In absolute terms, HapTree-X was able to phase a 200GB file in less than an hour as compared to the 22 hours needed by the fastest alternative tool [23] (note that Table 9.8 reports the measurements on a smaller subset of the original 120GB 10X BAM file; the results for the original dataset, as well as other datasets, are available in [23]). We should mention that HapTree-X is *not* a re-implementation of HapCUT2 or a similar tool, but a completely new tool

Haplotype phasing (HapTree-X)									
Linux	Time	Memory	Seq imprv.	Time	Memory	Seq imprv.	Time	Memory	Seq imprv.
HapCUT2 (C)	**1:23**	**0.1**	1.0×	**4:16**	**1.4**	0.7×	1:53:00	7.2	11.5×
phASER (Python)	1:53	0.7	1.3×						
HapTree-X (Seq)	1:25	2.9	—	5:47	1.0	—	**9:49**	**5.6**	—
macOS	Time	Memory	Seq imprv.	Time	Memory	Seq imprv.	Time	Memory	Seq imprv.
HapCUT2 (C)	1:29	**0.1**	1.7×	4:14	**1.1**	1.1×	1:29:19	7.9	12.1×
phASER (Python)	1:14	0.7	1.4×						
HapTree-X (Seq)	**52**	2.9	—	**3:47**	1.0	—	**7:24**	**5.9**	—
	(a) Small RNA-seq BAM (3 GB)			*(b) Medium WXS BAM (17 GB)*			*(c) Medium 10X BAM (13 GB)*		

Table 9.8: Performance improvements in haplotype phasing from using the Seq language and compiler on Linux and macOS systems. The Seq version achieves up to a 12× speed improvement over other tools.

that uses a different algorithm [23]; however, as all tools utilize many common data-processing steps that take the vast majority of time (SAM/BAM pre-processing, read analysis, etc.), it is fair to say that most of the speed improvements are due to Seq's optimizations.

9.2 Bioinformatics-Specific Benchmarks

We additionally evaluated the performance of Seq on the following three benchmark suites, designed to mimic both hypothetical and real-world genomics applications:

1. *The Computer Language Benchmarks Game* suite [53] restricted to DNA benchmarks (3 benchmarks)
2. Sequence manipulation suite, developed in-house (3 benchmarks)
3. Genomic index queries (2 benchmarks)

Benchmarking details We compared Seq with C++ (compiled with both GCC v7.4.0 and Clang v6.0.1), Julia v1.0.3, Python v2.7.15, PyPy v2.7.13 [27], Shed Skin v0.9.4 [47] and Nuitka v0.6.2 [56]. Other "compiled Python" implementations such as Numba are geared towards numerical rather string or DNA processing, and had issues efficiently compiling our benchmarks, or were abandoned. All experiments were run on a dual-socket system with Intel Xeon X5690 CPUs (3.46 GHz) with 6 cores each

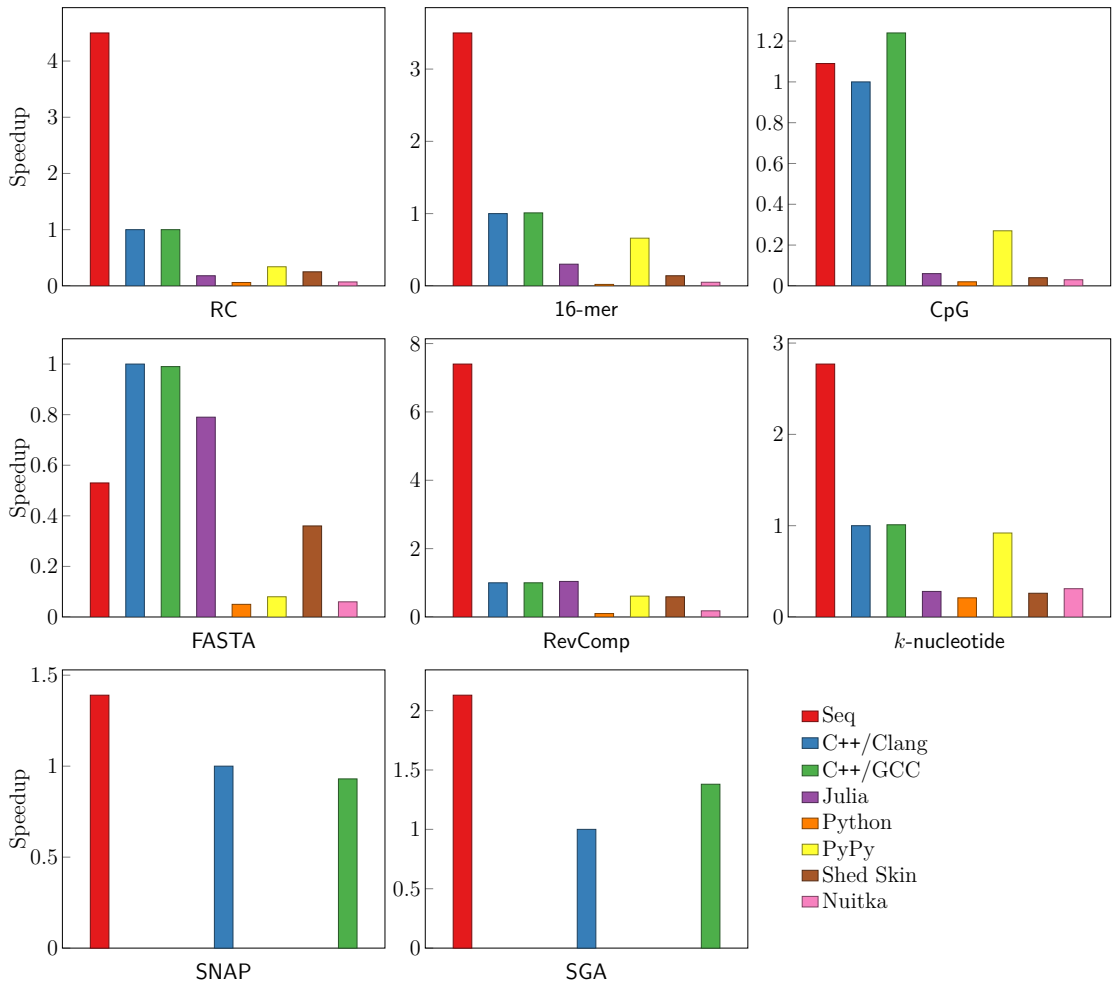


Figure 9-2: Seq evaluation results on several genomics benchmarks, showing speedups over Clang. The Seq implementations used in these charts use Seq-specific types and constructs that are not available in Python. Note that only Seq, Clang and GCC were tested on the SNAP and SGA benchmarks. Seq performs at least as good as (and in many cases much better than) the C++ implementations in nearly every benchmark, excluding FASTA which, as we note in the text, is not as common a real-world application as the others.

(totalling 12 cores and 24 hyper-threads) and 138GB DDR3-1333 RAM with 12MB LLC per socket. C++ implementations were compiled with `-O3 -march=native`. Julia was run with `--check-bounds=no -O3` parameters. Shed Skin was run with `-l -o` optimizations, and Nuitka was run with the `noasserts,no_warnings` options enabled. Note that Seq binaries (unlike C++ or Julia) do include bounds checks.

For the Benchmarks Game suite, we used the FASTA, RevComp and k-nucleotide microbenchmarks. Other benchmarks in this suite are not directly relevant to genomics or bioinformatics in general, but we expect Seq's performance on them to be on par with that of other LLVM-backed languages. Briefly, the FASTA benchmark entails generating random sequences in the FASTA format; RevComp entails reverse complementing a set of longer sequences, which is done through Seq's domain-specific sequence type; k-nucleotide entails counting k-mers of various lengths, which we implemented using Seq's k-mer types.

For the in-house suite, we designed three microbenchmarks that capture common genomics operations on a large set of reads:

1. RC: Output the reverse complement of each read, also implemented using sequence types. Unlike RevComp, this benchmark runs on millions of shorter reads rather than a few long reads.

2. 16-mer: Count the number of symmetric 16-mers in all reads (where a 16-mer is symmetric if its first half is identical to the reverse complement of its second half). The Seq implementation for this benchmark uses `match` on sequences, and is based on example (b) in Figure 3-7.

3. CpG: Count the number of CpG regions (i.e. regions that consist of C **and** G characters, such as `CGC` but *not* `CCC` or `GGG` as they lack a G and C, respectively) in all reads, and report the lengths of the shortest and longest CpG regions in the sample.

The final benchmark suite demonstrates the utility of Seq's domain-specific genomic

index query optimizations. Here, we use the genomic indices implemented in the widely-used tools SNAP v1.0 (beta 23) [129] (a sequence alignment tool) and SGA v0.10.15 [113] (a *de novo* assembly tool). SNAP uses a hash table of 20-mers, which we re-implemented from scratch in Seq (see Appendix B.1). SGA, on the other hand, uses an FM-index, whose C++ implementation we wrapped in Seq. Both indices are used to query k-mers from our test dataset. For both SNAP and SGA, we compared the performance of our base Seq implementation, a Seq version that performs pipeline optimizations for index prefetching (code difference of one line) and a C++ implementation (results shown at the bottom of Table 9.10). Both of these benchmarks consume roughly 30GB of RAM.

Each benchmark was executed five times for each language/compiler, and the averages are reported (with the exception of Julia, Python, Shed Skin, PyPy and Nuitka, which were run three times as they took orders of magnitude longer in some cases). The input dataset consisted of 100 million 75bp DNA reads randomly chosen from the HG00123 sample [1] (because SGA's index is an order of magnitude slower than SNAP's, we downsampled our input dataset to 25 million reads for SGA). Results are shown in Figure 9-2, where speedups over Clang-compiled C++ are given.

9.2.1 Improvements over Python

Seq programs can be written in one of two ways: in plain Python or using idiomatic Seq (as in the implementations described above). The Pythonic implementations utilize the coding conventions set by the Python community, and can be run by both Python and Seq directly. The alternative style involves the use of idiomatic Seq constructs and data types to manipulate genomic data; of course, these are not available in plain Python.

We compared the performance of Pythonic and idiomatic Seq implementations to that of Python, PyPy, Shed Skin, Nuitka and Julia in Table 9.9. All of the implementations in the second benchmark suite (with the exception of idiomatic Seq) are line-by-line identical in terms of the algorithm. In particular, the Python implementations use

Table 9.9: Seq runtime compared to Python, PyPy, Shed Skin, Nuitka and Julia (seconds). "Seq (Py.)" (Pythonic Seq) uses the same code as Python, whereas "Seq (Id.)" (idiomatic Seq) uses Seq-specific language features and constructs.

	Seq (Py.)	**Python**	**PyPy**	**Shed Skin**	**Nuitka**	**Julia**	**Seq (Id.)**	**Speedup**
FASTA		111.3	68.6	15.6	99.4	**7.1**	10.7	0.7–10×
RevComp		97.5	15.5	16.0	54.3	9.1	**1.3**	7–75×
k-nucleotide		255.8	59.2	211.8	174.1	196.0	**19.7**	10–13×
RC	195.1	2,160.0	231.2	912.1.6	913.7	764.7	**48.3**	5–45×
CpG	60.6	3,591.0	203.7	1,336.9	1,596.4	947.7	**60.6**	16–60×
16-mer	159.8	15,440.2	463.4	2,139.9	6,042.3	1,030.9	**95.6**	5–161×

the exact same code as the Pythonic Seq implementations for RC, CpG and 16-mer. Even by just directly running Python code, Seq is able to outperform Python by a factor of 11 to 100.

A similar pattern can be seen in the first benchmark suite, where Seq significantly outperforms both the Python and Julia implementations. The only exception is the FASTA benchmark, where Seq is slightly slower than Julia. While this could be further optimized, we chose to keep the version that is most similar to Python. Additionally, we note that the FASTA benchmark as specified by the Computer Language Benchmarks Game is not a realistic application in genomics, as one would rarely be generating sequences rather than reading them from a preexisting dataset.

Idiomatic versions further boost the improvement up to 160×, and showcase the impact of individual domain-specific optimizations: RC and RevComp utilize Seq's highly optimized reverse complementation constructs; 16-mer showcases the gains—both in terms of readability and performance—of sequence-based `match` statements; k-nucleotide shows the performance improvement gained by using Seq's native k-mer types. A few of the idiomatic versions also rely on pipelining to perform further optimizations, which is described in more detail below.

Note that runtime becomes prohibitive as the number of reads to be processed increases; while the performance of compiled Python (i.e. PyPy, Shed Skin and Nuitka) and Julia is acceptable if the number of reads is low (as in the first benchmark suite), it rapidly deteriorates once the read count becomes an order of magnitude larger.

Table 9.10: Seq runtime compared to C++ as compiled with Clang and GCC (seconds). "Pythonic Seq" uses the same code as Python, whereas "Idiomatic Seq" uses Seq-specific language features and constructs. For SNAP and SGA the difference between the without-prefetch and with-prefetch Seq programs is just a single `@prefetch` annotation.

	Seq Pythonic	**C++** Clang	**C++** GCC	**Seq** Idiomatic	**Speedup**
FASTA		**5.6**	5.7	10.7	0.5×
RevComp		9.5	9.5	**1.3**	7.3×
k-nucleotide		54.6	54.3	**19.7**	2.8×
RC	195.1	178.6	170.7	**48.3**	3.5×
CpG	60.6	55.7	**44.8**	60.6	0.7–0.9×
16-mer	159.8	214.1	201.7	**95.6**	2.1×

	Seq w/o prefetch	**C++** Clang	**C++** GCC	**Seq** with prefetch	**Speedup**
SNAP	328.1	450.5	327.5	**211.9**	1.5–2.1×
SGA	453.0	569.3	610.1	**409.6**	1.4–1.5×

Even 100 million reads as used here is quite minuscule compared to real datasets.

Given the results above, the comparisons below focus only on the C++ implementations.

9.2.2 Improvements over C++

Table 9.10 compares the Seq and C++ implementations of each benchmark. Again, all of these implementations (with the exception of idiomatic Seq) are line-by-line identical in terms of the algorithm. The performance of Pythonic Seq code is on par with that of C++ code—in most cases, it is the same as or slightly better than Clang's (we use Clang as a baseline since both Clang and Seq rely on LLVM for general-purpose optimizations). Note that g++ is sometimes able to outperform the LLVM-based backends of Seq and Clang. However, once Seq applies domain-specific optimizations, it outperforms even g++ by up to 4×. For example, in the third set of benchmarks (SNAP and SGA), Seq achieves a 50% speedup after adding a one-line domain-specific `prefetch` annotation to the original code, resulting in up to a 2×

Table 9.11: Seq runtime compared to that of several highly-optimized bioinformatics libraries (in seconds) on in-house benchmarks. C++ times are also included for reference.

	Seq	**C++** GCC	**SeqAn** C++/GCC	**BioPython** PyPy	**BioJulia** Julia
RC	**48.3**	170.7	137.6	68.4	348.9
CpG	60.6	**44.8**	46.7	332.7	244.1
16-mer	95.6	201.7	**70.8**	1,276.1	247.2

speedup over C++.

Prefetch variability

Index prefetching is useful during genomic index lookups, and is able to speed up both k-mer hash tables and FM-indices by 50%. However, we observed the performance of prefetching to be application- and data-dependent: while in almost all evaluated datasets (spanning various technologies such as recent third-generation 10x Genomics linked-reads [132] and "classic" second-generation Illumina short-reads; detailed results omitted for brevity) it produces a steady improvement in the range 20–50%, in one dataset it led to a 35% slowdown. However, the fact that it is a one-line change means that any user can easily experiment and judge whether it works well for their use-case.

Compilation time

Seq can be used in two modes: as a JIT interpreter or as a compiler. In our experiments, Seq's compilation times are faster than, or similar to, those of the C++ compilers. Note that Seq relies on LLVM's optimization pipeline, and therefore employs the same optimization passes (and linker) as Clang. We also observed that Seq is an order of magnitude faster than Nuitka or Shed Skin with respect to compilation.

Manual optimizations

In general, it is not straightforward to compare benchmark results across different languages in a meaningful way, given that each language has its own set of idioms

and conventions, often resulting in algorithmic differences. For example, if one invests enough time, it is always possible to write C++ implementations that match Seq's performance, since both ultimately compile to machine code. For this reason, the in-house benchmarks above all follow the same high-level algorithm, even if there may be room for further manual optimizations. The other two sets of benchmarks do include hand-optimized C++ implementations, however: FASTA, RevComp and k-nucleotide implementations are taken from The Computer Language Benchmarks Game (excluding multi-threaded implementations), and SNAP and SGA are real-world implementations that are widely used in practice.

We additionally compared to the highly-optimized bioinformatics libraries SeqAn v2.3.2 [45] (C++), BioPython v1.74 [38] and BioJulia (BioSequences v1.1.0) (`https://biojulia.net`), results for which are shown in Table 9.11. Seq outperforms both BioPython and BioJulia by a large margin. Seq also substantially outperforms SeqAn on RC; on CpG, the plain C++ implementation actually outperforms both; lastly, SeqAn outperforms Seq on 16-mer. Note that Seq matches SeqAn's performance if the same low-level implementation is used (the SeqAn implementation differs and is less flexible because Seq's sequence pattern matching cannot be expressed in SeqAn/C++). However, we purposefully compared against the (somewhat slower) pattern matching Seq implementation to show that even the "simplest" implementations in Seq are close performance-wise to the highly-optimized implementations provided by other libraries. Finally, these libraries are unable to easily express the benchmarks from the two other suites (and also lack a prefetching mechanism similar to what Seq uses in SNAP and SGA), which is why we limited this comparison to the in-house benchmarks.

9.2.3 Effects of parallelization

To evaluate the performance of Seq's parallel pipelines, we implemented parallel versions of two of our in-house benchmarks, CpG and 16-mer (the third, RC, performs substantially more I/O and hence does benefit much from parallelism), as well as

Table 9.12: Seq runtimes on multiple threads (seconds).

Threads	1	2	3	4	**Speedup**
CpG	58.1	29.6	19.9	15.3	3.8×
16-mer	86.7	43.6	29.9	22.8	3.8×
SGA	355.1	184.0	125.4	95.1	3.7×
SGA (pref.)	217.7	128.0	90.3	71.8	3.0×

SGA's FM-index querying (both with and without prefetch optimizations). To this end, we "block" input reads into batches of 100,000, which are processed as a whole by each task; tasks themselves are then executed in parallel via Seq's parallel pipe operator.

Results are shown in Table 9.12, where Seq scales almost linearly up to 4 threads on our in-house benchmarks. For these small applications, we find I/O to be a bottleneck beyond 4 threads. In a real-world setting where reads would take substantially longer to process, we would expect I/O to play a less significant role, allowing a greater degree of parallelism. Taking these parallel implementations into account, Seq's largest speedup over Python (which cannot be easily parallelized due to the global interpreter lock) is over 650×. Finally, Table 9.12 also shows that even Seq's prefetch optimizations benefit from parallelization.

9.3 General-Purpose Benchmarks

Because Seq can (with some restrictions) fully type check Python programs ahead of time, it can be used to compile regular Python code to native machine code even outside of genomics and bioinformatics. In Figure 9-3, we show Seq's advantage over standard CPython and PyPy (note that we have shown above that PyPy handily outperforms other implementations like Nuitka or Shed Skin, so we limit this comparison to PyPy for brevity). For most benchmarks, Seq is orders of magnitude faster. This difference is especially apparent for the loop benchmark, which iteratively calculates $\sum_{i=1}^{10^4-1}\sum_{j=1}^{10^4-1}(i+j)$; Seq's static typing and coroutine optimizations are able to optimize out the two loops entirely, replacing them with the result of the summation at

compile time.

Benchmarking details All benchmarks in this section were run on a dual-socket system with Intel Xeon E5-2695 v2 CPUs (2.40 GHz) with 12 cores each (totalling 24 cores and 48 hyper-threads) and 377GB DDR3-1066 RAM with 30MB LLC per socket, and Linux OS.

We selected several benchmark applications from Python's own benchmark suite[3], aside from loop which was taken from a PyPy tutorial[4], and is a simple double `for`-loop. We narrowed the benchmark suite by removing applications that relied on unsupported external libraries or modules (as they are not compatible with Seq natively, although they can be used in practice via Seq's out-of-the-box Python interoperability). Other benchmarks were preserved. A few of these benchmarks required slight modifications to be compatible with Seq's type system, such as specifying members of classes in advance (akin to Python's "data classes"), although we are currently working on extending LTS-DI to fully support such programs. The benchmark applications are as follows:

- loop: Simple double `for`-loop to compute $\sum_{i=1}^{10^4-1} \sum_{j=1}^{10^4-1} (i+j)$.
- norm (`bm_spectral_norm.py`): Spectral norm benchmark – calculates the spectral norm of an infinite matrix with specific entries; involves floating-point operations and list creation/iteration.
- nbody (`bm_nbody.py`): n-body simulation of several planets – repeatedly updates the momentum and position of each body in the system; involves list accesses, updates and iteration as well as floating-point arithmetic.
- float (`bm_float.py`): Floating point-heavy benchmark – optimizes a vector in 3-dimensional space based on some criteria; as the name suggests, primarily involves floating-point operations, as well as list iteration.

[3] `https://github.com/python/pyperformance`
[4] `https://realpython.com/pypy-faster-python`

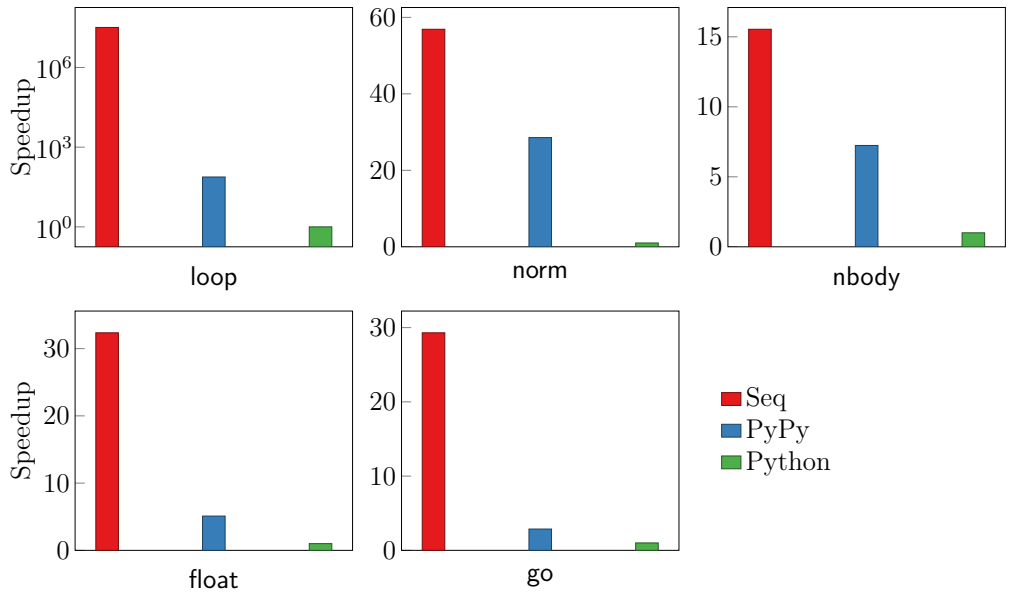


Figure 9-3: Comparison of Python (CPython 3), PyPy and Seq on several benchmarks from Python's benchmark suite (`https://github.com/python/pyperformance`).

- `go` (`bm_go.py`): AI for playing the Go board game – chooses a move in a Go game based on a tree search; involves tree construction and traversal as well as updating game state.

Similarly, Figure 9-5, shows the effects of the *get/set* optimization (discussed in Chapter 7) on a simple word-counting application (Figure 9-4). Note that the implementations for the Python, PyPy, and Seq versions are exactly identical. Nonetheless, the Seq version is twice as fast as Python and about 20% faster than C++. Enabling the get/set optimization results in a further 12% performance improvement.

9.4 Conclusion

This chapter showcases the collective impact of Seq's compiler optimizations on real-world, widely-used applications, as well as on a number of smaller benchmarks. Beyond just performance improvements, Seq affords programmers the ability to put aside low-level software design or performance considerations, and to simply focus on the high-level application and algorithm being implemented. Most of the C/C++

```cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <unordered_map>
using namespace std;

int main(int argc, char *argv[]) {
  cin.tie(nullptr);
  cout.sync_with_stdio(false);
  ifstream file(argv[1]);
  unordered_map<string, int> wc;
  for (string line; getline(file, line);) {
    istringstream sin(line);
    for (string word; sin >> word; )
      wc[word] += 1;
  }

  cout << wc.size() << endl;
}
```

```python
from sys import argv
wc = {}
filename = argv[1]

with open(filename) as f:
  for l in f:
    for w in l.split():
      wc[w] = wc.get(w, 0) + 1

print(len(wc))
```

Figure 9-4: C++ (left) and Python/Codon (right) implementations for a simple word counting program.

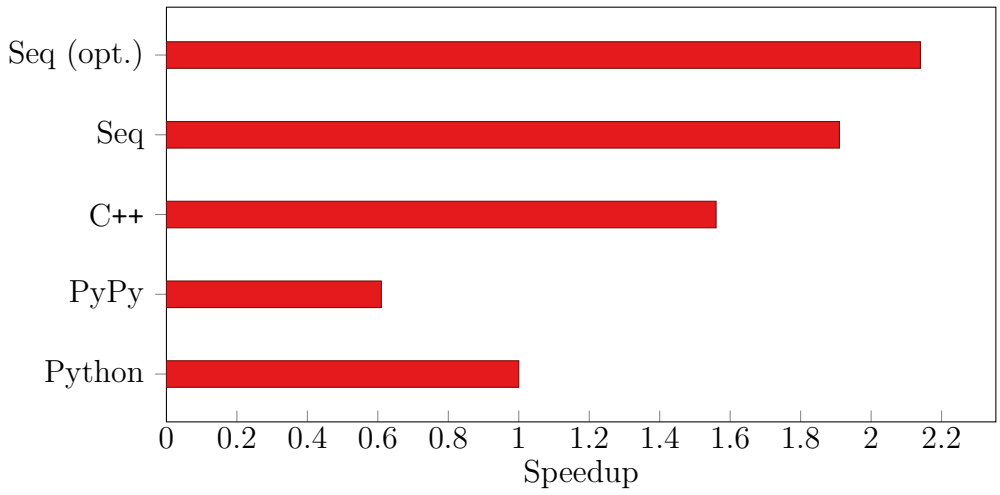


Figure 9-5: Runtimes for generating the word counts of 100 million lines of English Wikipedia text [51]. Speedups are relative to Python, and "Seq (opt.)" enables Seq's dictionary get/set IR optimization. The implementations are shown in Figure 9-4.

applications, in fact, were made several times shorter by Seq in terms of code size. Programming ease and simplicity has implications not only for initial development, but also for long-term maintenance, refactoring, and updating.

Chapter 10

Related Work

10.1 Genomics

A number of software libraries for bioinformatics and computational genomics have been developed for a variety of programming languages, such as Biopython [38] for Python, BioPerl [117] for Perl, SeqAn [45] for C++, Rust-Bio [65] for Rust or BioJulia [126] for Julia [24]. We have shown comparisons to most of these libraries in Chapter 9. Further, some of these libraries are bound to low-level host languages (C++, Rust), preventing inexperienced programmers from easily using them; others are bound to high-level languages that suffer from poor performance (Python, Perl). Languages like Julia attempt to bridge the gap between these extremes, but are infrequently used in practice [105]. Seq, by contrast, brings high-performance to a familiar, widely-used language (Python) that is among the most prevalent in the field. Furthermore, many of the optimizations performed by the Seq compiler are simply infeasible for libraries to replicate; for example, the inter-sequence alignment optimization described in Chapter 6 requires a global view of the program, the ability to generate arbitrary code, the ability to convert functions to coroutines at compile time and so on—these transformations are inherently out of reach for libraries, and require a domain-specific compiler. Consequently, libraries are unable to match Seq's

performance on many of the applications re-implemented in this work.

Another line of work focuses on integrating various high-level code blocks into a pipeline framework that can be efficiently run on large clusters and cloud-based systems—examples include the Broad Institute's HAIL project and Workflow Description Language [123]. While these methods indeed allow for large-scale parallelism and a relatively high-level description of a given problem, they are cumbersome to use as they require rather expensive infrastructural setup and administration costs, and do not tackle the problem of single-machine optimizations, which is still a significant bottleneck in many pipelines.

Seq is inspired by many successful DSLs that already exist in other fields of computer science [101, 3, 14, 131, 34, 36, 62, 63]. Despite their substantial success in these other areas, computational biology has yet to adopt a comparable DSL. SARVAVID [81] is a DSL designed for computational genomics applications, which provides a set of high-level genomics kernels and exposes them as language constructs. For example, common operations such as `k-merization`, `index-generation`, `index-lookup`, `similarity-computation` and `clustering` are provided. While such an approach provides efficient implementations of these kernels and combinations thereof, it lacks generality, which is gravely needed in the field as new sequencing technologies produce new types of data that in turn necessitate novel algorithms. Seq aims to provide a more general, lower-level language, with general-purpose constructs that can be used to build a variety of kernels efficiently. SARVAVID is also not available online—neither source code nor compiled binaries—and is not used in practice.

10.2 Type Checking

Much work has been done on applying type inference to various dynamic languages in general, and to Python in particular. Here is a short, and by no means complete, survey of past work:

Mypy The most popular type checker for Python is arguably mypy[1]. Mypy is more of a linter or a static analysis tool than a traditional type checker: it is completely orthogonal to the Python runtime, its usage is optional, and it does not impact the performance of a program. It attempts to cover the whole language in all its intricacies, and performs a decent job of spotting common typing pitfalls in Python code. As Python code often cannot be fully analyzed by static analysis tools, mypy is forced to leave some types ambiguous (represented as an `Any` type). Mypy also requires manual type annotations in certain complicated scenarios, since it cannot infer complex polymorphic types automatically. Other similar type checkers and linters include Pylance[2] and Pytype[3].

RPython RPython [9, 103] is a restricted subset of the Python language that aims to be amenable to static analysis tools. As RPython can be statically analyzed, it can also be unambiguously typed. RPython's type system is quite different from ML-based systems: RPython's "annotator" (the key typing module) computes program flow graphs, and then continually propagates types through flow graphs in a top-down fashion. This approach is inspired by older work that focuses on localized type inference of atomic types [32]. While some aspects of RPython are highly similar to LTS-DI (e.g. some of the core language restrictions and its localized nature), RPython does not use an ML-like system, and the supported syntax is more restricted than what LTS-DI supports (with the exception of the inheritance, which LTS-DI currently does not support). The overall goal of the project is also different: RPython is not intended to be used in a standalone manner, but as a building block for creating dynamic interpreters. It currently serves as the key component of the JIT translator in the PyPy project [26]. PyPy itself is a full re-implementation of the Python language and uses RPython only when it is possible to do so.

[1] `http://mypy-lang.org`
[2] `https://marketplace.visualstudio.com/items?itemName=ms-python.vscode-pylance`
[3] `https://github.com/google/pytype`

Starkiller Starkiller [106] is another static type inference method that utilizes the Cartesian Product Algorithm [5] for type inference. Starkiller is flow-insensitive and does not claim to support all the intricacies of Python's semantics. Similar to LTS-DI, Starkiller relies heavily on monomorphization—the technique of instantiating new functions for different input arguments—to simulate Python's duck typing. However, it does not support exceptions or generators and is furthermore not available anymore. It also targeted a much earlier Python version as it was developed before 2004.

Python compilers Projects such as Cython [22], PyPy [26], Numba [66], Shed Skin [47] and Nuitka [56] all aim to generate efficient code by relying on ideas such as static type checking and (JIT-) compiling rather than interpreting Python. Most of these implementations, however, either still rely on the Python runtime, and are thus bound to its inherent performance overhead, or support a small subset of Python (e.g. Numba is only applicable to functions that perform numerical computations). Other similar implementations (e.g. Pythran, Pyston, Grumpy) are either geared more towards numerical computing or are no longer under active development.

Dynamic-like languages Initial versions of Seq utilized a simple uni-directional type system akin to C++'s. This approach is also shared by Crystal [82], a static, compiled version of Ruby. In such a system, each type must be decided at once, and expressions are not allowed to have an undecided type at any point once they have been seen during type checking. While this approach is both simple and efficient, it is not powerful enough to cover much of Python's syntax and requires excessive type annotations, especially for optional and container types—both staples of any Python codebase. Furthermore, it does not allow for lambda functions nor several other syntactic elements prevalent in Python code.

Non-Python type systems Many type systems were designed for other dynamic languages such as Ruby, including InferDL [59], Hummingbird [102] and PRuby [52]. Another notable example is TypeScript[4]. These approaches often rely on heuristics

[4] `https://www.typescriptlang.org/`

such as matching variable names to class types, approximations which are then modelled as constraints and used to infer types. While this results in good coverage of the language, these approaches have many of the same disadvantages as mypy and other Pythonic type checkers—they are essentially auxiliary to the program and allowed to fail. LTS-DI, on the other hand, necessitates a complete view of all types in a program in order to facilitate the generation of native machine code and does not aim to cover every feature of the host language.

10.3 Intermediate Representations

While Seq IR is not the first customizable IR, it differs from frameworks like MLIR [68] in its approach. Rather than support customization of *everything*, Seq's IR offers a clearly-defined, restricted way for DSL writers to add new features. This improves compatibility and makes Seq a fully batteries-included framework. In terms of structure, SIR takes inspiration from LLVM and Rust's IRs [121]. These IRs benefit from a vastly simplified node set and structure, which in turn simplifies the implementation of IR passes. Structurally, however, these representations are too low-level to effectively express many optimizations used by Seq. In particular, they radically restructure the source code, eliminating semantic information that must be reconstructed to perform transformations. To address this shortcoming, many IRs like Taichi [58] and Suif [127] adopt hierarchical structures that maintain control-flow and semantic information, albeit at the cost of increased complexity. Unlike Seq's, however, these IRs are largely disconnected from their languages' front-ends, making maintaining type correctness and generating new code impractical or even impossible. Therefore, SIR takes the best of these approaches by utilizing a *simplified* hierarchical structure, maintaining both the source's control flow nodes and a radically decreased subset of internal nodes. Importantly, it augments this structure with bidirectionality, making new IR easy to generate and manipulate.

10.4 Extensible Compilers

Many frameworks for implementing DSLs rely on "embedding" into another language and extending its features with metaprogramming techniques. Delite [31] is one such tool, relying on Scala's rich operator overloading support to implement custom syntax. The framework represents applications as valid Scala code, and constructs an intermediate representation at runtime; DSL optimizations can manipulate this IR through "rewrite rules" and "traversals" before final compilation. AnyDSL [70] takes a similar approach of embedding in a new language called Impala. While these tools can achieve considerable customization, they are necessarily limited by their parent languages in terms of syntax and typing flexibility. Further, since both derive from strongly-typed languages and do not tie in with their host type system, they are unable to reach the level of expressiveness provided by dynamic languages. Other frameworks like Sham [125], based on Racket, embed in dynamic languages and emit IR. However, these approaches are difficult to work with in practice and rely on languages that are either new themselves or not widely used, limiting adoption. By contrast, Codon tackles the DSL problem in the context of an existing, widely-used dynamic language, which comes with its own unique challenges that are not directly addressed by prior work.

Chapter 11

Conclusion

The creation of high-performance, maintainable bioinformatics software is undoubtedly a difficult task, as evidenced by the field's unfortunate track record of poor software development, maintenance, and testing practices. A key cause of this, among other factors, is a lack of domain-specific development tooling, debugging facilities and general programming infrastructure. While several software libraries have been developed as an attempt at answering this problem, libraries are unable to perform many of the optimizations discussed in this work and hence cannot achieve optimal performance in many key situations. Additionally, these libraries are still bound to their host languages, usually at the detriment of either performance or software maintainability.

The fundamental difference with Seq is that its entire software stack, from the language itself to the standard library to the compiler and even runtime, is designed and implemented with the domain of bioinformatics in mind. "Owning" the entire stack in this way comes with numerous distinct advantages. For example, datatypes or compiler optimizations relevant to new sequencing platforms can be incorporated seamlessly into the framework, circumventing large code rewrites or potentially even algorithmic changes. Additionally, as GPUs, FPGAs, and cloud computing become increasingly prevalent in the field, new compiler backends for Seq—including new

compiler optimizations pertinent to each computing platform—would enable existing code to run without changes on these backends, and is ongoing work within the Seq project. Indeed, many novel, promising computer architectures such as memory-driven computing [21] require specific optimizations and program transformations to be fully taken advantage of; these can be performed automatically by the Seq compiler.

Yet another unique advantage of a framework like Seq lies in its ability to provide domain-specific debugging or visualization support; as the compiler has knowledge of sequences, k-mers and operations involving them, debugging or visualization tools can be seamlessly integrated into the Seq framework, allowing users to track and observe in a meaningful way how data is transferred and manipulated throughout a program. None of these possibilities are easily attainable without having a software stack designed from the ground up for genomics and bioinformatics. In fact, such domain-specific languages are widely used in a variety of other fields for similar reasons, including computer graphics [101], tensor algebra computing [62] and physical simulations [63].

11.1 Future Work

The work presented here serves as a foundation for bioinformatics-specific compilers, languages, and optimizations, but ultimately still only scratches the surface of what is possible. There are many exciting avenues for further research and exploration, including:

- *New compiler backends.* In this thesis, we primarily focused on CPU-based computing, but as mentioned above, bioinformatics and many other fields alike are increasingly moving towards new computing platforms like GPUs, cloud computing, or even FPGAs. Each of these provides unique opportunities and challenges in terms of compilation, optimization, and execution.

- *New compiler optimizations.* There are a number of interesting domain-specific

optimizations that we have yet to explore. For example, can the compiler deduce the best way to encode sequences based on context, be it an ASCII encoding, 2-bit, or something else? Can we further exploit the unique, non-uniform structure of the genome to attain better performance? Can we use subtle properties of sequences, like error profiles, to our advantage in certain situations? Each of these questions is an exciting research direction and warrants further investigation.

- *Full Python compatibility.* Seq is able to statically compile a large subset of Python, but there is still substantial room to close the gap even further. For example, it would be possible to use union types to allow for some of the cases that Seq currently cannot handle, as well as to extend the system to support dynamic polymorphism.

11.2 Closing Remarks

Seq is, to the best of our knowledge, the first programming language tailored for bioinformatics that combines high-performance and scalability with a familiar, high-level, and easy-to-maintain programming system or language. It is the first step towards a rich ecosystem providing software development environments, visualization tools, and debugging support, all designed with the domain of bioinformatics in mind. By offering biologists, bioinformaticians and other researchers a scalable way to prototype, experiment and analyze large biological datasets through a familiar, high-performance language, we hope that Seq will act as a catalyst for scientific discovery and innovation.

Appendix A

Seq Tutorial

Here we will demonstrate the capabilities of Seq by implementing a short and fast read alignment pipeline, named SeqMap, with only 35 lines of high-level Python code. Further details, together with the links to the test datasets, can be found at `https://docs.seq-lang.org/tutorial/workshop`.

We will test SeqMap on the following data:

1. Chromosome 22 as the reference sequence (`chr22.fa`), and
2. A simulated set of 1 million 101bp reads (`reads.fastq`).

We assume that Seq has been properly installed and configured; for more details on how to do this, please visit `https://seq-lang.org`. We also assume that a reader is familiar with the basic bioinformatics formats (FASTA, FASTQ, etc.).

A.1 Reading Sequences from Disk

The first step of processing any kind of sequencing data is to read it from disk. Seq has builtin support for many of the standard file formats in bioinformatics, such as FASTA, FASTQ, SAM, BAM and CRAM.

Let's write a program to read our FASTQ file and print each record's name and sequence on a single line:

```
from sys import argv
from bio import *
for record in FASTQ(argv[1]):
    print record.name, record.seq
```

Now we can run this Seq program as follows:

```
seqc run section1.seq data/reads.fq > out.txt
```

and peek into the `out.txt`:

```
chr22_16993648_16994131_1:0:0_2:0:0_0/1 CTACCAAACACCTACTTCGTTTCCTAACATCACTTTAATTTTATCTTAGAGGAATTCTTTTCCCTATCCCATTAAGTTATGGGAGATGGGGCCAGGCATGG
chr22_28253010_28253558_1:0:0_0:0:0_1/1 AGTGTTTTGCCTGTGGCTAGACTAAAAATAAGGAATGAGGGGGGTATCTTCCACTCTTGCCCTCTCATCACCCTATTCCCTATATCCAGAACTCAGAGTCC
chr22_21656192_21656802_0:0:0_2:0:0_2/1 ATAGCGTGGATTCCTATGACATCAAGGAGCTATTTTATTTGGTAAAACGAAAAAGCACAATAATGAACGAACGCAAGCACTGAAACAGTGGAGACACCTAG
chr22_44541236_44541725_0:1:0_0:0:0_3/1 CTCTCTGTCTCTCTCTCTCCCCTAGGTCAGGGTGGTCCCTGGGGAGGCCCCTGGGTTACCCCAAGACAGGTGGGAGGTGCTTCCTACCCGACCCTCTTCCT
chr22_39607671_39608139_0:0:0_2:0:0_4/1 ATTGGCTCAGAGTTCAGCAGGCTGTACCAGCATGGCGCCAGTGTCTGCTCCTGGTGAGGCCTTACGGACGTTACAATAACGGCGGAAGGCAAAGGCGGAGC
chr22_35577703_35578255_3:0:0_1:0:0_5/1 TGCCATGGTGGTTAGCTGCACCCATCAACCTGTCATCTACATTAGGTATTTTTCCTAATGCTATCCCTCCCCTAGCACCCTACCCTCTGATAGGCCCTGGT
chr22_46059124_46059578_1:0:0_1:0:0_6/1 AATCAGTACCAAACAATATATGGATATTATTGGCACTTTGTGCTCCCTCTGCCTGAACTGGGAATTCCTCTATTAGTTTTGACATTATCTGGTATTGAACC
chr22_31651867_31652385_2:0:0_2:0:0_7/1 ATCTAGTGACAGTAAGTGGCTGATAAAGTGAGCTGCCATTACATAGTCATCATCTTTAATAGAAGTTAACACATACTGAGTTTCTACTATATTGGGTCTTT
chr22_24816466_24817026_1:0:0_1:0:0_8/1 CACCTCTAGGGCTCAAGGGGCAGTTCCTCCATTCCTCAGCAGTGGCGCCTGTGGAACTGTGTCCTGAGGCCAGGGGGTGGTCAGGCAGGGCCTGGAGTGGC
chr22_27496272_27496752_1:0:0_1:0:0_9/1 CTTAGCCCCATTAAACACTGGCAGGGCTGAATTGTCTGCTGCCATCCATCACACCTTCTCCCCTAGCCTGGTTTCTTACCTACCTGGAAGCCGTCCCTTTT
...
```

That was it! FASTA, BAM and other file formats can be read in a very similar way. Note that since this data is simulated, each read's name (e.g. `chr22_16993648_...`) indicates the locus at which the read was obtained.

A.2 Building an Index

Our goal is to find a "mapping" to the genome for each read in a FASTQ file. Comparing to every position on the reference sequence would take far too long. An alternative is to create an index of the k-mers from the reference sequence and use it to guide the mapping process [128].

Let's build a dictionary that links each 32-mer to its position (*locus*) on the reference sequence:

```
K = Kmer[32]  # make a type alias to improve the readability
index = {}
for record in FASTA(argv[1]):
    for pos,kmer in record.seq.kmers_with_pos[K](step=1):
        index[kmer] = pos
```

Of course, there will be k-mers that appear multiple times, but let's ignore this detail for now and just store the latest position we see for each k-mer.

Another important issue is reverse complementation: some of our reads will map in the reverse direction rather than in the forward direction. For this reason, we will build our index in such a way that a k-mer is considered "equal" to its reverse complement. One easy way to do this is by using *canonical k-mers*, i.e. the minimum of a k-mer and its reverse complement. For example, the canonical form of `GCT` is its reverse complement, `AGC`, since the reverse complement is lexicographically smaller (i.e. comes first alphabetically). On the other hand, the canonical form of `CGC` is itself, since it is lexicographically smaller than its reverse complement, `GCG`. Here is the new code:

```
index = {}
for record in FASTA(argv[1]):
    for pos, kmer in record.seq.kmers_with_pos[K](step=1):
        index[min(kmer, ~kmer)] = pos
```

Note that we will have to use canonical k-mers when querying the index as well.

Now we have our index as a dictionary (`index`), but we do not want to build it each time we perform read mapping, since it only depends on a (fixed) reference sequence. So, as a last step, let's dump the index to a file using the `pickle` module:

```
import pickle, gzip
with gzip.open(argv[1] + '.index', 'wb') as jar:
    pickle.dump(index, jar)
```

Let us run the program from the command line:

```
seqc run section2.seq data/chr22.fa
```

We should now be able to see a new file `data/chr22.fa.index` which stores our serialized index.

Note that thus far, Seq both feels and behaves like a typical Python program.

A.3 Finding Seed Matches

At this point, we have an index we can load from disk. We will use it to find candidate mappings for our reads. Let us first split each read into k-mers, and report a mapping if at least two k-mers "support" a particular locus.

The first step is to load the index:

```
index = None
with gzip.open(argv[1] + '.index', 'rb') as jar:
    # pickle needs to know the original structure type
    index = pickle.load[Dict[K,int]](jar)
```

Now we can iterate over our reads and query k-mers in the index. We need a way to keep track of candidate mapping positions as we process the k-mers of a read: we can do this using a new dictionary `candidates` that maps candidate alignment positions to the number of k-mers supporting the given position [128]. Then, we just iterate over candidates and output positions supported by 2 or more k-mers. Finally, we clear candidates before processing the next read:

```
candidates = {}  # map position to a count
for record in FASTQ(argv[2]):
    for pos, kmer in record.read.kmers_with_pos[K](step=1):
        found = index.get(min(kmer, ~kmer), -1)
        if found > 0:
```

```
            candidates.increment(found - pos)
    for pos, count in candidates.items():
        if count > 1:
            print record.name, pos + 1
    candidates.clear()
```

Let us run the program and peek at the output:

```
seqc run section3.seq data/chr22.fa data/reads.fq | head
chr22_16993648_16994131_1:0:0_2:0:0_0/1 16993648
chr22_28253010_28253558_1:0:0_0:0:0_1/1 28253010
chr22_44541236_44541725_0:1:0_0:0:0_3/1 44541236
chr22_31651867_31652385_2:0:0_2:0:0_7/1 31651867
chr22_21584577_21585142_1:0:0_1:0:0_a/1 21584577
chr22_46629499_46629977_0:0:0_2:0:0_b/1 47088563
chr22_46629499_46629977_0:0:0_2:0:0_b/1 51103174
chr22_46629499_46629977_0:0:0_2:0:0_b/1 46795988
chr22_16269615_16270134_0:0:0_1:0:0_c/1 50577316
chr22_16269615_16270134_0:0:0_1:0:0_c/1 16269615
```

Notice that most positions we reported match the position from the simulated read name (the first integer after the _); not bad for such a short program!

A.4 Smith-Waterman Alignment and CIGAR String Generation

We now have the ability to report approximate mapping positions for each read. However, we usually need more precise alignments, which include information about mismatches, insertions and deletions. Luckily, Seq makes sequence alignment easy: to align sequence q against sequence t, you can just do:

```
aln = q @ t
```

`aln` is a tuple of alignment score and CIGAR string (a CIGAR string is a way of encoding an alignment result, and consists of operations such as `M` for match/mismatch, `I` for insertion and `D` for deletion, accompanied by the number of associated bases; for example, `3M2I4M` indicates 3 (mis)matches followed by a length-2 insertion followed by 4 (mis)matches).

By default, Levenshtein distance is used, meaning mismatch and gap costs are both 1, while match costs are zero. More control over alignment parameters can be achieved using the `align` method:

```
aln = q.align(t, a=2, b=4, ambig=0, gapo=4, gape=2)
```

where `a` is the match score, `b` is the mismatch cost, `ambig` is the ambiguous base (N) match score, `gapo` is the gap open cost, and `gape` the gap extension cost (i.e. a gap of length k costs $\texttt{gapo} + (k \cdot \texttt{gape})$). There are many more parameters as well, controlling factors like alignment bandwidth, Z-drop, global/extension alignment and more; please check the standard library reference for further details.

To align a read to the reference, we will use a simple `query.align(target)`:

```
candidates = {}
for record in FASTQ(argv[2]):
    for pos, kmer in record.read.kmers_with_pos[K](step=1):
        found = index.get(min(kmer, ~kmer), -1)
        if found > 0:
            candidates.increment(found - pos)
    for pos, count in candidates.items():
        if count > 1:
            # get query, target and align:
            query = record.read
            target = reference[pos:pos + len(query)]
            alignment = query.align(target)
            print record.name, pos + 1, alignment.score, alignment.cigar
```

```
    candidates.clear()
```

Now, run the program and observe the output:

```
seqc run section4.seq data/chr22.fa data/reads.fq | head
chr22_16993648_16994131_1:0:0_2:0:0_0/1 16993648 196 101M
chr22_28253010_28253558_1:0:0_0:0:0_1/1 28253010 196 101M
chr22_44541236_44541725_0:1:0_0:0:0_3/1 44541236 196 101M
chr22_31651867_31652385_2:0:0_2:0:0_7/1 31651867 190 101M
chr22_21584577_21585142_1:0:0_1:0:0_a/1 21584577 196 101M
chr22_46629499_46629977_0:0:0_2:0:0_b/1 47088563 110 20M1I4M1D76M
chr22_46629499_46629977_0:0:0_2:0:0_b/1 51103174 134 20M1I4M1D76M
chr22_46629499_46629977_0:0:0_2:0:0_b/1 46795988 128 20M1I4M1D76M
chr22_16269615_16270134_0:0:0_1:0:0_c/1 50577316 118 101M
chr22_16269615_16270134_0:0:0_1:0:0_c/1 16269615 202 101M
```

Most of the alignments contain only matches or mismatches (`M`), which is to be expected as insertions and deletions are far less common. In fact, the three mappings containing indels appear to be incorrect!

A more thorough mapping scheme would also look at alignment scores before reporting mappings, although for the purposes of this tutorial we will ignore such improvements.

A.5 Pipelines

Pipelines are a very convenient Seq construct for expressing a variety of algorithms and applications. In fact, SeqMap can be thought of as a pipeline with the following stages:

1. read a record from the FASTQ file,

2. find candidate alignments by querying the index, and

3. perform alignment for mappings supported by 2+ k-mers and output results.

We can write these stages as separate functions in Seq as follows:

```
def find_candidates(record):
    candidates = {}
    for pos, kmer in record.read.kmers_with_pos[K](step=1):
        found = index.get(min(kmer, ~kmer), -1)
        if found > 0:
            candidates.increment(found - pos)
    for pos, count in candidates.items():
        if count > 1:
            yield record, pos
def align_and_output(t):
    record, pos = t
    query = record.read
    target = reference[pos:pos + len(query)]
    alignment = query.align(target)
    print record.name, pos + 1, alignment.score, alignment.cigar
```

and then chain them into a pipeline:

```
FASTQ(argv[2]) |> iter |> find_candidates |> align_and_output
```

Notice that `find_candidates` yields candidate alignments to `align_and_output`, which then performs alignment and prints the results. In Seq, all values generated from one stage of a pipeline are lazily passed to the next stage. Lazy passing allows the Seq pipeline to behave like Bash streams: no new item will be generated until the current item is processed. The Seq compiler performs many domain-specific optimizations on pipelines, one of which we focus on next.

A.5.1 Parallelism

Pipelines can be easily parallelized with the parallel pipe operator, `||>`, which tells the compiler that all subsequent stages can be executed in parallel:

```
FASTQ(argv[2]) |> iter ||> find_candidates |> align_and_output
```

Since the full program also involves loading the index, let us time the main pipeline using the `timing` module to see if parallelism helps or not:

```
import timing
with timing('mapping'):
    FASTQ(argv[2]) |> iter ||> find_candidates |> align_and_output
```

Let us run this for different numbers of threads (e.g., 1 and 2):

```
export OMP_NUM_THREADS=1  # this sets the number of threads Seq can use
seqc run section5.seq data/chr22.fa data/reads.fq > out.txt
# mapping took 48.2858s

export OMP_NUM_THREADS=2
seqc run section5.seq data/chr22.fa data/reads.fq > out.txt
# mapping took 35.886s
```

Often, batching reads into larger blocks and processing those blocks in parallel can yield better performance, especially if each read is quick to process. This is also very easy to do in Seq:

```
def process_block(block):
    block |> iter |> find_candidates |> align_and_output
with timing('mapping'):
    FASTQ(argv[2]) |> blocks(size=2000) ||> process_block
```

The effect of batching is clearly visible now:

```
export OMP_NUM_THREADS=1
```

```
seqc run section5.seq data/chr22.fa data/reads.fq > out.txt
# mapping took 48.2858s

export OMP_NUM_THREADS=2
seqc run section5.seq data/chr22.fa data/reads.fq > out.txt
# mapping took 25.2648s
```

A.6 Domain-Specific Optimizations

Seq already performs numerous domain-specific optimizations under the hood. However, we can give the compiler a hint in this case to perform one more: inter-sequence alignment (see Supplementary Note 3 for more details). This optimization entails batching sequences prior to alignment, then aligning multiple pairs using a very fast vectorized (SIMD-optimized) alignment kernel.

In Seq, we just need one additional function annotation to tell the compiler to perform this optimization:

```
@inter_align
def align_and_output(t):
    ...
```

Let us run the program with and without this optimization:

```
# without @inter_align
seqc run section5.seq data/chr22.fa data/reads.fq > out.txt
# mapping took 43.4457s

# with @inter_align
seqc run section6.seq data/chr22.fa data/reads.fq > out.txt
# mapping took 32.3241s
```

The timings with inter-sequence alignment will depend on the SIMD instruction sets

your CPU supports; we ran our experiments on an AVX2-compatible machine.

A.7 Final Code Listing

Here is the final 35-line long code listing for our toy mapper. Note that achieving the same result in general-purpose languages—particularly with inter-sequence alignment—would require several hundred or potentially thousands of lines of code.

```
# Usage: seqc run section6.seq <FASTA path> <FASTQ path>
from sys import argv
from time import timing
from bio import *
import pickle
import gzip

K = Kmer[32]
index = None
reference = s''
for record in FASTA(argv[1]):  # Take the first sequence
    reference = record.seq
with gzip.open(argv[1] + '.index', 'rb') as jar:  # Load the index
    index = pickle.load[Dict[K, int]](jar)

def find_candidates(record):
    candidates = {}
    for pos, kmer in record.read.kmers_with_pos[K](step=1):
        found = index.get(min(kmer, ~kmer), -1)
        if found > 0:
            candidates.increment(found - pos)
    for pos, count in candidates.items():
        if count > 1:
            yield record, pos
```

```
25
26 @inter_align
27 def align_and_output(t):
28     record, pos = t
29     query = record.read
30     target = reference[pos:pos + len(query)]
31     alignment = query.align(target)
32     print record.name, pos + 1, alignment.score, alignment.cigar
33
34 with timing('mapping'):
35     FASTQ(argv[2]) |> iter |> find_candidates |> align_and_output
```

A.8 Using Non-Seq Libraries

There are cases where you might want to access some Python (or C) library that is not yet ported to Seq. For example, you might want to use Biopython [38] to parse a rare or non-standard file format. To allow seamless integration with existing ecosystems, Seq supports running pure Python code, as well as using Python libraries and objects directly within Seq code.

For example, suppose you need to analyze the output of a NCBI BLAST run, and extract all BLAST hits within a given e-value threshold. You can use the following code to achieve this, and then use the data from BLAST directly in Seq:

```
from python import Bio.Blast.NCBIXML

# Everything within a @python function will be executed by the Python runtime
@python
def process_blast(file, threshold):
    from Bio.Blast import NCBIXML
    result_handle = open(file)
    for record in NCBIXML.parse(result_handle):
```

```
        for alignment in record.alignments:
            for hsp in alignment.hsps:
                if hsp.expect < threshold:
                    yield (alignment, hsp)

hits = process_blast('blast.xml', 0.04)
for aln, hsp in hits:
    # Use Python objects directly ...
    print aln.accession, 'of length', aln.length
    # ... or cast them to Seq objects for improved performance:
    print str(hsp.query)[:100]
    print str(hsp.sbjct)[:100]
    e_value = hsp.expect.get[float]()
    print e_value
```

For more details, please consult the official Seq documentation at `https://docs.seq-lang.org`.

Appendix B

Selected Code Listings

B.1 Code from SNAP Benchmark

SNAP uses a hierarchical hash table as its genomic index. To index k-mers ($k \geq 16$), an array of 4^{k-16} quadratic probing hash tables is created, indexed by every possible length-$(k-16)$ prefix. Each constituent hash table is then a mapping of 16-mer to genomic loci at which that 16-mer appears. To handle multiple loci for a single k-mer, an auxiliary array is used, and hash table values can be pointers into this array (determined by whether the value is greater than the largest locus). This hierarchical structure exploits the fact that not every length-$(k-16)$ prefix appears in the genome with equal frequency, so the size of each internal hash table can be chosen based on the corresponding prefix's frequency.

SNAP's index is implemented in C++, and can be found at `https://github.com/amplab/snap`. Below, we give the Seq implementation used in the SNAP benchmark. The code that queries this table largely resembles what is shown in Figure 3-1 (both for Seq and C++). Note that loading the precomputed table from disk is still done in C++, and wrapped in Seq.

```
# File: hashtable.seq
# Implementation of SNAP aligner's hash table
# https://github.com/amplab/snap/blob/master/SNAPLib/HashTable.{cpp,h}
QUADRATIC_CHAINING_DEPTH = 5
class SNAPHashTable[K,V]:
    table: array[tuple[V,K]]
    invalid_val: V

    def _hash(k):
        key = hash(k)
        key ^= int(u64(key) >> u64(33))
        key *= 0xff51afd7ed558ccd
        key ^= int(u64(key) >> u64(33))
        key *= 0xc4ceb9fe1a85ec53
        key ^= int(u64(key) >> u64(33))
        return key

    def __init__(self: SNAPHashTable[K,V], size: int, invalid_val: V):
        self.table = array[tuple[V,K]](size)
        self.invalid_val = invalid_val
        for i in range(size):
            self.table[i] = (invalid_val, K())

    def __init__(self: SNAPHashTable[K,V], p: ptr[byte]):
        cdef snap_hashtable_ptr(ptr[byte]) -> ptr[tuple[V,K]]
        cdef snap_hashtable_len(ptr[byte]) -> int
        cdef snap_hashtable_invalid_val(ptr[byte]) -> V
        self.table = array[tuple[V,K]](snap_hashtable_ptr(p), snap_hashtable_len(p))
        self.invalid_val = snap_hashtable_invalid_val(p)

    def _get_index(self: SNAPHashTable[K,V], where: int):
        return int(u64(where) % u64(len(self.table)))

    def get_value_ptr_for_key(self: SNAPHashTable[K,V], k: K):
        table = self.table
        table_size = table.len
        table_index = self._get_index(SNAPHashTable[K,V]._hash(k))
        invalid_val = self.invalid_val
        entry = table[table_index]
        if entry[1] == k and entry[0] != invalid_val:
            return ptr[V](table.ptr + table_index)
        else:
            n_probes = 0
            while True:
                n_probes += 1
                if n_probes > table_size + QUADRATIC_CHAINING_DEPTH:
                    return ptr[V]()
                diff = (n_probes**2) if n_probes < QUADRATIC_CHAINING_DEPTH else 1
                table_index = (table_index + diff) % table_size
                entry = table[table_index]
                if not (entry[1] != k and entry[0] != invalid_val):
                    break
            return ptr[V](table.ptr + table_index)

    def __prefetch__(self: SNAPHashTable[K,V], k: K):
        table = self.table
        table_index = self._get_index(SNAPHashTable[K,V]._hash(k))
        (self.table.ptr + table_index).__prefetch_r3__()

    def __getitem__(self: SNAPHashTable[K,V], k: K):
        p = self.get_value_ptr_for_key(k)
        return p[0] if p else self.invalid_val
```

```
# File: genomeindex.seq
# Implementation of SNAP aligner's genome index
# https://github.com/amplab/snap/blob/master/SNAPLib/GenomeIndex.{cpp,h}
from hashtable import SNAPHashTable
type k16 = Kmer[16]

class GenomeIndex[K]:
    hash_tables: array[SNAPHashTable[k16,u32]]
    overflow_table: array[u32]
    count_of_bases: int

    def _partition(k: K):
        n = int(k.as_int())
        return (k16(n & ((1 << 32) - 1)), n >> 32)

    def __init__(self: GenomeIndex[K], dir: str):
        assert 16 <= K.len() <= 32
        cdef snap_index_from_dir(ptr[byte]) -> ptr[byte]
        cdef snap_index_ht_count(ptr[byte]) -> int
        cdef snap_index_ht_get(ptr[byte], int) -> ptr[byte]
        cdef snap_index_overflow_ptr(ptr[byte]) -> ptr[u32]
        cdef snap_index_overflow_len(ptr[byte]) -> int
        cdef snap_index_count_of_bases(ptr[byte]) -> int

        p = snap_index_from_dir(dir.c_str())
        assert p
        hash_tables = array[SNAPHashTable[k16,u32]](snap_index_ht_count(p))
        for i in range(len(hash_tables)):
            hash_tables[i] = SNAPHashTable[k16,u32](snap_index_ht_get(p, i))

        self.hash_tables = hash_tables
        self.overflow_table = array[u32](snap_index_overflow_ptr(p), snap_index_overflow_len(p))
        self.count_of_bases = snap_index_count_of_bases(p)

    def __getitem__(self: GenomeIndex[K], seed: K):
        kmer, which = GenomeIndex[K]._partition(seed)
        table = self.hash_tables[which]
        value_ptr = table.get_value_ptr_for_key(kmer)

        if not value_ptr or value_ptr[0] == table.invalid_val:
            return array[u32](value_ptr, 0)

        value = value_ptr[0]

        if int(value) < self.count_of_bases:
            return array[u32](value_ptr, 1)
        else:
            overflow_table_offset = int(value) - self.count_of_bases
            hit_count = int(self.overflow_table[overflow_table_offset])
            return array[u32](self.overflow_table.ptr + overflow_table_offset + 1, hit_count)

    def __prefetch__(self: GenomeIndex[K], seed: K):
        kmer, which = GenomeIndex[K]._partition(seed)
        table = self.hash_tables[which]
        table.__prefetch__(kmer)
```

```
# File: snap.seq
# k-mer counting using SNAP's hash table
from genomeindex import *
from sys import argv
type K = Kmer[20]
good = 0
bad = 0

@prefetch
def process(kmer: K, index: GenomeIndex[K]):
    global good, bad
    hits = index[kmer]
    hits_rc = index[~kmer]

    if len(hits) > 0 or len(hits_rc) > 0:
        good += 1
    else:
        bad += 1

assert len(argv) == 3
index = GenomeIndex[K](argv[1])
step = 10
fastq(argv[2]) |> kmers[K](step) |> process(index)
print good, bad
```

```cpp
// File: snap.cpp
// k-mer counting using SNAP's hash table

#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cstdint>
#include <cassert>
using namespace std;

extern "C" void *snap_index_from_dir(char *dir);
extern "C" void snap_index_lookup(void *idx, char *bases, int64_t *nHits, const unsigned **hits,
                                  int64_t *nRCHits, const unsigned **rcHits);

// filter ambiguous bases from sequences
static bool hasN(char *kmer, unsigned len)
{
        for (unsigned i = 0; i < len; i++) {
                if (kmer[i] == 'N')
                        return true;
        }
        return false;
}

int good = 0;
int bad = 0;

int main(int argc, char *argv[])
{
        assert(argc == 3);
        void *idx = snap_index_from_dir(argv[1]);
        const unsigned k = 20;
        const unsigned step = 10;
        unsigned hit = 0;
        unsigned hit_rc = 0;
        int64_t n_hits = 0;
        int64_t n_hits_rc = 0;
        const unsigned *hit_p = &hit;
        const unsigned *hit_rc_p = &hit_rc;
        ifstream fin(argv[2]);
        string read;
        long line = -1;

        while (getline(fin, read)) {
                line++;
                if (line % 4 != 1) continue;  // skip non-sequences in FASTQ
                unsigned max_pos = 0, max_count = 0;
                char *buf = (char *)read.c_str();
                unsigned len = read.size();

                for (unsigned i = 0; i + k <= len; i += step) {
                        if (hasN(&buf[i], k)) continue;
                        snap_index_lookup(idx, &buf[i], &n_hits, &hit_p, &n_hits_rc, &hit_rc_p);
                        ++((n_hits > 0 || n_hits_rc > 0) ? good : bad);
                }
        }

        cout << good << " " << bad << endl;
}
```

B.2 Code from SMEMs Benchmark

Below is the SMEM-finding algorithm from BWA-MEM, implemented in Seq. The `@prefetch` annotation tells the compiler to use software prefetching to accelerate the FM-index queries performed in this function, leading to a 2× performance improvement.

```
@prefetch
def fastmap(rec: FASTQRecord, fmi: FMIndex, out: File):
    prev, curr, mems = list[SMEM](), list[SMEM](), list[SMEM]()
    q, l, start = rec.seq, len(q), 0
    while True:
        while start < l and q[start].N(): start += 1
        if start >= l: break
        mems.clear()
        prev.clear()
        curr.clear()
        x = start
        if q[x].N(): return
        ik = SMEM(fmi.biinterval(q[x]), start=x, stop=x+1)

        # forward search
        i = x + 1
        while i < l:
            if not q[i].N():  # an A/C/G/T base
                ok = ~fmi[~(ik.interval), ~q[i]]
                if len(ok) != len(ik.interval):  # change of the interval size
                    curr.append(ik)
                    if len(ok) < min_intv:
                        break  # the interval size is too small to be extended further
                ik = SMEM(ok, start=x, stop=i+1)
            else:  # an ambiguous base
                curr.append(ik)
                break
            i += 1

        if i == l:
            curr.append(ik)
        curr.reverse()
        ret = curr[0].stop
        prev, curr = curr, prev

        # backward search for MEMs
        i = x - 1
        while i >= -1:
            c = i >= 0 and not q[i].N()
            curr.clear()
            for p in prev:
                ok = FMDInterval()
                if c:
                    ok = fmi[p.interval, q[i]]
                if not c or len(ok) < min_intv:
                    if len(curr) == 0:
                        if len(mems) == 0 or i + 1 < mems[-1].start:
                            ik = SMEM(p.interval, start=i+1, stop=p.stop)
                            if len(ik) >= min_seed:
                                mems.append(ik)
                elif len(curr) == 0 or len(ok) != len(curr[-1].interval):
                    curr.append(SMEM(ok, start=p.start, stop=p.stop))
            if len(curr) == 0:
                break
            prev, curr = curr, prev
            i -= 1

        mems.reverse()  # s.t. sorted by the start coordinate
        start = ret
        output(mems)
```

Appendix C

Seq Reference Guide

C.1 Seq Standard Library

This section outlines Seq's `bio` library as of v0.10. Seq also supports many standard Python modules, which abide by Python's standard library documentation.

- `bio.align`: Functions for Smith-Waterman alignment, including global and extension alignment with many parameters including affine gap scores, dual gap scores, Z-drop, bandwidth and more. This module also includes functions relevant to inter-sequence alignment, including the coroutine scheduler, dispatcher, flusher, and the data structures for managing state.

- `bio.bam`: Utilities for reading BAM, SAM and CRAM files. Seq interfaces with HTSlib [28] for reading several common file formats.

- `bio.bed`: Utilities for reading BED files.

- `bio.block`: Utilities for pipeline or I/O "blocking", which entails batching elements into larger blocks. Blocking is particularly effective when running parallel code and when each data element is quick to process—in this case, blocking amortizes the thread management cost over many data elements.

- `bio.builtin`: Collection of commonly-used bio functions, including functions like `revcomp` (reverse complement), `split` (subsequence iteration), `kmers` (k-merization) and many more.

- `bio.bwa`: Interface to external BWA alignment tool [73], allowing the use of BWA within a Seq program to align sequences to a reference.

- `bio.bwt`: Low-level implementation of the "Suffix Array by Induced Sorting" (SA-IS) algorithm for linear-time SA construction [91]. This module also includes methods for building the SA, Burrows-Wheeler transform (BWT), and longest common prefix (LCP) array for DNA and protein sequences.

- `bio.c_htslib`: C bindings for HTSlib.

- `bio.fai`: Utilities for reading FAI ("FASTA index") files. FAI files serve as indices for much larger FASTA files.

- `bio.fasta`: Utilities for reading FASTA files optionally accompanied by an indexing FAI file. The module provides options for validation, compressed I/O, and sequence copying.

- `bio.fastq`: Utilities for reading FASTQ files. The module provides options for validation, compressed I/O, and sequence copying.

- `bio.fmindex`: Implementation of FM-index and FMD-index data structures. These data structures can either be constructed from a given sequence object or directly from a FASTA file.

- `bio.intervals`: Utilities for working with genomic intervals. Includes a suffix array implementation for storing sets of intervals, finding intersections, and so on.

- `bio.iter`: Utilities for iterating over sequences from a file, with one sequence per line.

- `bio.locus`: Standardized genomic loci. Representing a genomic locus is often cumbersome due to the discrepancy between 0-based and 1-based indexing, the need to maintain chromosome names, and other miscellaneous considerations—this module offers a standardized representation of genomic loci that also accounts for reverse strands.
- `bio.pseq`: Implementation of the `pseq` type for storing protein sequences, as well as many methods for operating on them.
- `bio.seq`: Implementation of the `seq` type for storing DNA/RNA sequences, as well as many methods for operating on them.
- `bio.vcf`: Utilities for reading VCF files.

In-depth documentation is available at `https://docs.seq-lang.org/stdlib/bio`.

C.2 Seq vs. Python – A Cheat Sheet

C.2.1 Additional types

- `seq`: Represents a genomic sequence.
- `Kmer[`N`]` ($1 \leq N \leq 1024$): Represents a k-mer of length N. N must be constant.
- `Int[`N`]` ($1 \leq N \leq 2048$): Represents a signed N-bit integer (standard `int` is an `Int[64]`). N must be constant. The common type definitions `i8`, `i16`, `i32` and `i64` are provided in the standard library for convenience.
- `UInt[`N`]` ($1 \leq N \leq 2048$): Represents an unsigned N-bit integer. N must be constant. The common type definitions `u8`, `u16`, `u32` and `u64` are provided in the standard library for convenience.
- `Ptr[T]`: Represents a pointer to type `T`; primarily useful for C interoperability.

- `Array[T]`: Represents an array of type `T` (essentially a pointer with a length).

C.2.2 Additional keywords and annotations

- `@type`: Indicates that a given class should be treated as a named tuple.
- `@extend`: Adds the given methods to an existing type.
- `@prefetch`: Indicates that the annotated function should be subjected to the prefetch optimization (Chapter 6).
- `@inter_align`: Indicates that the annotated function should be subjected to the inter-sequence alignment optimization (Chapter 6).
- `match`/`case`: Match statement (note that upcoming Python versions will include structural pattern matching as implemented in Seq; refer to `https://www.python.org/dev/peps/pep-0622` for additional information).
- `__ptr__`: Obtains a pointer to the given variable (similar to taking the address of a local variable in C).
- `__array__`: Declares a stack-allocated, fixed-size array.

Additional information can be found at `https://docs.seq-lang.org/tutorial`.

C.2.3 Static types

Because Seq is statically-typed, lists (for example) cannot contain elements of different types as they can in Python. Similarly, a variable cannot be assigned to objects of different types, nor can a function return objects of different types. Seq currently also does not support polymorphism.

C.2.4 Tuples

Tuples in Seq are implemented as structures. Consequently, heterogeneous tuples can only be indexed by a constant value, since otherwise the type of the index expression would be ambiguous. (Iteration over heterogeneous tuples is made possible by unrolling the loop at compile time, though.)

C.2.5 Scopes

Seq enforces slightly stricter variable scoping rules than standard Python. In short, a variable cannot be first assigned in a block then used afterwards for the first time in the enclosing block. This restriction avoids the problem of uninitialized variables.

Appendix D

Seq AST and IR Listings

D.1 AST Nodes

Node	Python Equivalent	De-sugared to?
`PassStmt`	`pass`	✗
`BreakStmt`	`break`	✗
`ContinueStmt`	`continue`	✗
`ExprStmt`	Expression.	✗
`AssignStmt`	Variable assignment.	✗
`UpdateStmt`	Variable update.	✗
`DelStmt`	`del`	✗
`PrintStmt`	`print`	Function call.
`ReturnStmt`	`return`	✗
`YieldStmt`	`yield`	✗
`AssertStmt`	`assert`	Function call.
`ImportStmt`	`import`	Removed.
`ThrowStmt`	`throw`	✗
`GlobalStmt`	`global`	Removed.
`YieldFromStmt`	`yield from`	✗

Table D.1: Listing of simple AST statements.

Node	Python Equivalent	De-sugared to?
`SuiteStmt`	Block of statements.	✗
`WhileStmt`	`while` loop	✗
`ForStmt`	`for` loop	✗
`IfStmt`	`if/else` block	✗
`MatchStmt`	n/a	If statements.
`TryStmt`	`try/catch` block	✗
`FunctionStmt`	Function declaration.	Function.
`ClassStmt`	Class declaration.	Type and flattened functions.
`WithStmt`	`with` block.	✗
`CustomStmt`	n/a	DSL-specific.

Table D.2: Listing of complex AST statements.

Node	Python Equivalent	De-sugared to?
`NoneExpr`	`None`	Function call.
`BoolExpr`	`bool` literal.	✗
`IntExpr`	Integer literal.	✗
`FloatExpr`	Float literal.	✗
`StringExpr`	`str` literal.	✗
`IdExpr`	Identifier.	✗
`StarExpr`	Unpack expression.	Function argument.
`KeywordStarExpr`	Keyword arg star.	Function argument.
`TupleExpr`	Tuple literal.	Tuple construction.
`ListExpr`	List literal.	List construction.
`SetExpr`	Set literal.	Sect construction.
`DictExpr`	Dictionary literal.	Dictionary construction.
`GeneratorExpr`	Comprehension.	Construction and append.
`DictGeneratorExpr`	Dictionary comprehension.	Construction and append.
`IfExpr`	Ternary operation.	✗
`UnaryExpr`	Unary expression.	Function call.
`BinaryExpr`	Binary expression.	Function call.
`ChainBinaryExpr`	Range comparison expression.	Function call.
`PipeExpr`	n/a	✗
`IndexExpr`	Index expression.	Function call.
`CallExpr`	Call expression.	✗
`DotExpr`	Member access expression.	✗
`SliceExpr`	Slice expression.	Function call.
`EllipsisExpr`	n/a	Partial call.
`TypeOfExpr`	Type-of expression.	✗
`LambdaExpr`	Lambda function expression.	Function.
`YieldExpr`	Yield-in expression.	✗
`AssignExpr`	n/a	✗
`RangeExpr`	n/a	Function call.
`StmtExpr`	n/a	✗
`PtrExpr`	n/a	✗
`TupleIndexExpr`	Tuple index expression.	✗
`StackAllocExpr`	n/a	✗

Table D.3: Listing of AST expressions.

D.2 SIR Nodes

	Description	LLVM Equivalent
`IntType`	64-bit integer.	`i64`
`FloatType`	64-bit float.	`double`
`BoolType`	Boolean.	`i8`
`ByteType`	8-bit integer.	`i8`
`VoidType`	Void.	`void`
`RecordType`	Struct.	`StructType`
`RefType`	Pointer to a struct.	`PointerType`
`FuncType`	Function type.	`FunctionType`
`PointerType`	Pointer.	`PointerType`
`OptionalType`	Optional value.	`PointerType`
`GeneratorType`	Generator.	`PointerType`
`IntNType`	Variable length integer.	`i{N}`

Table D.4: Listing of builtin SIR types.

	Type	Description
`Var`	Variable	Global or local variable.
`BodiedFunc`	Function	SIR function.
`LLVMFunc`	Function	Function implemented in LLVM IR.
`ExternalFunc`	Function	Function implemented in library.
`InternalFunc`	Function	Function implemented in compiler.

Table D.5: Listing of SIR variables.

	Type	Description
`IntConst`	Constant	Integer value.
`FloatConst`	Constant	Float value.
`BoolConst`	Constant	Boolean value.
`StrConst`	Constant	String value.
`AssignInstr`	Instruction	Sets a variable's value.
`ExtractInstr`	Instruction	Gets a member.
`InsertInstr`	Instruction	Sets a member.
`CallInstr`	Instruction	Calls a function.
`StackAllocInstr`	Instruction	Allocates an array.
`TypePropertyInstr`	Instruction	Checks a type property.
`YieldInInstr`	Instruction	Gets a value yielded in.
`TernaryInstr`	Instruction	Ternary operator.
`BreakInstr`	Instruction	Breaks a loop.
`ContinueInstr`	Instruction	Continues a loop.
`ReturnInstr`	Instruction	Returns from a function.
`YieldInstr`	Instruction	Yields from a function.
`ThrowInstr`	Instruction	Throws an exception.
`FlowInstr`	Instruction	Executes a flow.
`SeriesFlow`	Flow	Basic block flow.
`ForFlow`	Flow	For loop.
`WhileFlow`	Flow	While loop.
`IfFlow`	Flow	Conditional flow.
`IfFlow`	Flow	Conditional flow.
`TryCatchFlow`	Flow	Exception handling flow.
`PipelineFlow`	Flow	Pipeline flow.

Table D.6: Listing of SIR values.

Bibliography

[1] 1000 Genomes Project Consortium, Gonçalo R. Abecasis, David Altshuler, Adam Auton, Lisa D. Brooks, Richard M. Durbin, Richard A. Gibbs, Matt E. Hurles, and Gil A. McVean. A Map of Human Genome Variation from Population-Scale Sequencing. *Nature*, 467(7319):1061–1073, October 2010.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[4] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, 2(1):53–86, 2004.

[5] Ole Agesen. The cartesian product algorithm. In *European Conference on Object-Oriented Programming*, pages 2–26. Springer, 1995.

[6] Ziad Al Bkhetan, Justin Zobel, Adam Kowalczyk, Karin Verspoor, and Benjamin Goudey. Exploring effective approaches for haplotype block phasing. *BMC Bioinformatics*, 20(1):540, Oct 2019.

[7] Erdoğan Aldemir et al. Binary medical image compression using the volumetric run-length approach. *The Imaging Science Journal*, 67(3):123–135, 2019.

[8] Anaconda. Numba, 2018.

[9] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. Rpython: A step towards reconciling dynamically and statically typed oo languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, page 53–64, New York, NY, USA, 2007. Association for Computing Machinery.

[10] R. Appuswamy, J. Fellay, and N. Chaturvedi. Sequence alignment through the looking glass. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 257–266, May 2018.

[11] John Aycock. Aggressive type inference. In *Proceedings of the 8th International Python Conference*, pages 11–20, 2000.

[12] Stefan Milton Bache and Hadley Wickham. magrittr: A forward-pipe operator for r. *R package version*, 1(1), 2014.

[13] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 193–205. IEEE Press, 2019.

[14] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 193–205, Piscataway, NJ, USA, 2019. IEEE Press.

[15] J. A. Bailey, A. M. Yavor, H. F. Massa, B. J. Trask, and E. E. Eichler. Segmental duplications: organization and impact within the current human genome project assembly. *Genome Research*, 11(6):1005–1017, 2001.

[16] Jeffrey A Bailey, Amy M Yavor, Hillary F Massa, Barbara J Trask, and Evan E Eichler. Segmental duplications: organization and impact within the current human genome project assembly. *Genome research*, 11(6):1005–1017, 2001.

[17] Monya Baker. 1,500 scientists lift the lid on reproducibility. *Nature News*, 533(7604):452, 2016.

[18] S Batzoglou, L Pachter, JP Mesirov, B Berger, and ES Lander. Human and mouse gene structure: comparative analysis and application to exon prediction. *Genome research*, 10(7):950—958, July 2000.

[19] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 420–432, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[20] David Beazley. Understanding the python gil. In *PyCON Python Conference. Atlanta, Georgia*, 2010.

[21] Matthias Becker, Hartmut Schultze, Kirk Bresniker, Sharad Singhal, Thomas Ulas, and Joachim L. Schultze. A novel computational architecture for large-scale genomics. *Nature Biotechnology*, Sep 2020.

[22] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, 2011.

[23] Emily Berger, Deniz Yorukoglu, Lillian Zhang, Sarah K. Nyquist, Alex K. Shalek, Manolis Kellis, Ibrahim Numanagić, and Bonnie Berger. Improved haplotype inference by exploiting long-range linking and allelic imbalance in rna-seq datasets. *Nature Communications*, 11(1):4662, Sep 2020.

[24] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv*, page 1209.5145, 2012.

[25] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, September 1988.

[26] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[27] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[28] James K Bonfield, John Marshall, Petr Danecek, Heng Li, Valeriu Ohan, Andrew Whitwham, Thomas Keane, and Robert M Davies. HTSlib: C library for reading/writing high-throughput sequencing data. *GigaScience*, 10(2), 02 2021. giab007.

[29] Nick Bray, Inna Dubchak, and Lior Pachter. Avid: A global alignment program. *Genome research*, 13(1):97–102, 2003.

[30] Broad Institute. *Picard Tools*. `http://broadinstitute.github.io/picard/`.

[31] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 89–100, 2011.

[32] Brett Cannon. Localized type inference of atomic types in python, 2005.

[33] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *Financial Cryptography and Data Security*, pages 35–50, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[34] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. *SIGPLAN Not.*, 46(8):35–46, February 2011.

[35] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3), August 2007.

[36] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. Diderot: a parallel dsl for image analysis and visualization. In *Acm sigplan notices*, volume 47, pages 111–120. ACM, 2012.

[37] Hyunghoon Cho, David J. Wu, and Bonnie Berger. Secure genome-wide association analysis using multiparty computation. *Nature Biotechnology*, 36(6):547–551, Jul 2018.

[38] Peter JA Cock, Tiago Antao, Jeffrey T Chang, Brad A Chapman, Cymon J Cox, Andrew Dalke, Iddo Friedberg, Thomas Hamelryck, Frank Kauff, Bartek Wilczynski, et al. Biopython: freely available python tools for computational molecular biology and bioinformatics. *Bioinformatics*, 25(11):1422–1423, 2009.

[39] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2012.

[40] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4(1-10):1–8, 2001.

[41] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

[42] Morten Dahl, Chao Ning, and Tomas Toft. On secure two-party integer division. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security*, pages 164–178, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[43] Jeff Daily. Parasail: Simd c library for global, semi-global, and local pairwise sequence alignments. *BMC Bioinformatics*, 17(1):81, 2016.

[44] Luis Damas. Type assignment in programming languages. *KB thesis scanning project 2015*, 1984.

[45] Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. Seqan an efficient, generic c++ library for sequence analysis. *BMC Bioinformatics*, 9(1):11, 2008.

[46] Erwan Drezen, Guillaume Rizk, Rayan Chikhi, Charles Deltel, Claire Lemaitre, Pierre Peterlongo, and Dominique Lavenier. Gatb: Genome assembly & analysis tool box. *Bioinformatics (Oxford, England)*, 30(20):2959–2961, Oct 2014. 24990603[pmid].

[47] Mark Dufour. Shed skin: An optimizing python-to-c++ compiler. Master's thesis, Delft University of Technology, 2006.

[48] Michael Farrar. Striped Smith–Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, 11 2006.

[49] Paolo Ferragina and Giovanni Manzini. Compression boosting in optimal linear time using the Burrows-Wheeler Transform. In *SODA 2004*, pages 655–663, 2004.

[50] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122, January 2004.

[51] Wikimedia Foundation. Wikimedia downloads, 2021.

[52] Michael Furr, Jong-hoon An, and Jeffrey S Foster. Profile-guided static typing for dynamic scripting languages. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 283–300, 2009.

[53] Isaac Gouy. The computer language benchmarks game. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`.

[54] Faraz Hach, Iman Sarrafi, Farhad Hormozdiari, Can Alkan, Evan E. Eichler, and S. Cenk Sahinalp. mrsFAST-Ultra: A compact, SNP-aware mapper for high performance sequencing applications. *Nucleic Acids Research*, 42(W1):W494–W500, 2014. `http://nar.oxfordjournals.org/content/42/W1/W494.abstract`.

[55] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez

del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[56] K Hayen. Nuitka, 2012.

[57] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[58] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: A language for high-performance computation on spatially sparse data structures. *ACM Trans. Graph.*, 38(6), November 2019.

[59] Milod Kazerounian, Brianna M. Ren, and Jeffrey S. Foster. Sound, heuristic type annotation inference for ruby. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2020, page 112–125, New York, NY, USA, 2020. Association for Computing Machinery.

[60] Abdul Rafay Khan, Muhammad Tariq Pervez, Masroor Ellahi Babar, Nasir Naveed, and Muhammad Shoaib. A comprehensive study of de novo genome assemblers: Current challenges and future prospective. *Evol Bioinform Online*, 14:1176934318758650–1176934318758650, Feb 2018. 29511353[pmid].

[61] Vladimir Kiriansky, Haoran Xu, Martin Rinard, and Saman Amarasinghe. Cimple: Instruction and memory level parallelism: A dsl for uncovering ilp and mlp. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, pages 30:1–30:16, New York, NY, USA, 2018. ACM.

[62] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. Taco: A tool to generate tensor algebra kernels. In *Proc. IEEE/ACM Automated Software Engineering*, pages 943–948. IEEE, 2017.

[63] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David IW Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M Kaufman, Gurtej Kanwar, Wojciech Matusik, et al. Simit: A language for physical simulation. *ACM Transactions on Graphics (TOG)*, 35(2):20, 2016.

[64] Gregory Kucherov, Karel Břinda, and Maciej Sykulski. Spaced seeds improve k-mer-based metagenomic classification. *Bioinformatics*, 31(22):3584–3592, 07 2015.

[65] Johannes Köster. Rust-Bio: a fast and safe bioinformatics library. *Bioinformatics*, 32(3):444–446, 10 2015.

[66] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM.

[67] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, Palo Alto, California, 2004.

[68] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law, 2020.

[69] Robyn S Lee and William P Hanage. Reproducibility in science: important or incremental? *The Lancet Microbe*, 2020.

[70] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. Anydsl: A partial evaluation framework for programming high-performance libraries. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[71] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[72] Heng Li. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, 05 2012.

[73] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem, 2013.

[74] Heng Li. Minimap2: fast pairwise alignment for long dna sequences. *arXiv preprint arXiv:1708.01492*, 2017.

[75] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[76] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[77] Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Brief Bioinform*, 11(5):473–483, Sep 2010. 20460430[pmid].

[78] Jan-Yie Liang, Chih-Sheng Chen, Chua-Huang Huang, and Li Liu. Lossless compression of medical images using hilbert space-filling curves. *Computerized Medical Imaging and Graphics*, 32(3):174–182, 2008.

[79] Hengyun Lu, Francesca Giordano, and Zemin Ning. Oxford nanopore minion sequencing and genome assembly. *Genomics, Proteomics & Bioinformatics*, 14(5):265–279, 2016.

[80] George F Luger. *Artificial intelligence: structures and strategies for complex problem solving*. Pearson education, 2005.

[81] Kanak Mahadik, Christopher Wright, Jinyi Zhang, Milind Kulkarni, Saurabh Bagchi, and Somali Chaterji. Sarvavid: A domain specific language for developing scalable computational genomics applications. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 34:1–34:12, New York, NY, USA, 2016. ACM.

[82] Manas. Crystal.

[83] Swati C Manekar and Shailesh R Sathe. A benchmark study of k-mer counting methods for high-throughput sequencing. *GigaScience*, 7(12), 10 2018. giy125.

[84] Teri A Manolio, Lisa D Brooks, and Francis S Collins. A hapmap harvest of insights into the genetics of common disease. *The Journal of Clinical Investigation*, 118(5):1590–1605, 2008.

[85] ER Mardis. DNA sequencing technologies: 2006-2016. *Nature Protocols*, 12(2):213–218, 2017.

[86] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 01 2011.

[87] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, and Mark A. DePristo. The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 20(9):1297–1303, September 2010.

[88] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[89] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[90] Gor Nishanov. Iso/iec ts 22277:2017, Dec 2017.

[91] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *2009 Data Compression Conference*, pages 193–202, 2009.

[92] Ibrahim Numanagić, Alim S Gökkaya, Lillian Zhang, Bonnie Berger, Can Alkan, and Faraz Hach. Fast characterization of segmental duplications in genome assemblies. *Bioinformatics*, 34(17):i706–i714, 2018.

[93] Takeshi Ogasawara, Yinhe Cheng, and Tzy-Hwa Kathy Tzeng. Sam2bam: High-performance framework for NGS data preprocessing tools. *PloS one*, 11(11):e0167100, 2016.

[94] Brian D. Ondov, Todd J. Treangen, Páll Melsted, Adam B. Mallonee, Nicholas H. Bergman, Sergey Koren, and Adam M. Phillippy. Mash: fast genome and metagenome distance estimation using minhash. *Genome Biology*, 17(1):132, Jun 2016.

[95] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[96] William R Pearson and David J Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988.

[97] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[98] Roger D Peng. Reproducible research in computational science. *Science*, 334(6060):1226–1227, 2011.

[99] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[100] Pypl popularity of programming language index.

[101] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[102] Brianna M Ren and Jeffrey S Foster. Just-in-time static type checking for dynamic languages. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 462–476, 2016.

[103] A Rigo, M Hudson, and S Pedroni. Compiling dynamic language implementations, ist fp6-004779.

[104] Torbjørn Rognes. Faster smith-waterman database searches with inter-sequence simd parallelisation. *BMC Bioinformatics*, 12(1):221, 2011.

[105] Pamela H. Russell, Rachel L. Johnson, Shreyas Ananthan, Benjamin Harnke, and Nichole E. Carlson. A large-scale analysis of bioinformatics code on github. *PLOS ONE*, 13(10):1–19, 10 2018.

[106] Michael Salib. *Starkiller: A static type inferencer and compiler for Python.* PhD thesis, Massachusetts Institute of Technology, 2004.

[107] R. R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, 1997.

[108] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. *SIGPLAN Not.*, 52(8):249–265, January 2017.

[109] Ariya Shajii, Ibrahim Numanagić, Riyadh Baghdadi, Bonnie Berger, and Saman Amarasinghe. Seq: A high-performance language for bioinformatics. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[110] Ariya Shajii, Ibrahim Numanagić, Riyadh Baghdadi, Bonnie Berger, and Saman Amarasinghe. Seq: A high-performance language for bioinformatics. *Proc. ACM Program. Lang.*, 3(OOPSLA):125:1–125:29, October 2019.

[111] Ariya Shajii, Ibrahim Numanagić, Alexander T. Leighton, Haley Greenyer, Saman Amarasinghe, and Bonnie Berger. A python-based programming language for high-performance computational genomics. *Nature Biotechnology*, Jul 2021.

[112] Ariya Shajii, Ibrahim Numanagić, Christopher Whelan, and Bonnie Berger. Statistical binning for barcoded reads improves downstream analyses. *Cell Systems*, 7(2):219–226, 2018.

[113] Jared T. Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Res*, 22(3):549–556, Mar 2012. 22156294[pmid].

[114] Petr Šmarda, Petr Bureš, Lucie Horová, Ilia J. Leitch, Ladislav Mucina, Ettore Pacini, Lubomír Tichý, Vít Grulich, and Olga Rotreklová. Ecological and evolutionary significance of genomic gc content diversity in monocots. *Proceedings of the National Academy of Sciences*, 111(39):E4096–E4102, 2014.

[115] Geoffrey S Smith. Polymorphic type inference with overloading and subtyping. In *Colloquium on Trees in Algebra and Programming*, pages 671–685. Springer, 1993.

[116] Tom Sean Smith, Andreas Heger, and Ian Sudbery. Umi-tools: Modelling sequencing errors in unique molecular identifiers to improve quantification accuracy. *Genome Research*, 2017.

[117] Jason E. Stajich, David Block, Kris Boulez, Steven E. Brenner, Stephen A. Chervitz, Chris Dagdigian, Georg Fuellen, James G. R. Gilbert, Ian Korf, Hilmar Lapp, Heikki Lehväslaiho, Chad Matsalla, Chris J. Mungall, Brian I. Osborne, Matthew R. Pocock, Peter Schattner, Martin Senger, Lincoln D. Stein, Elia Stupka, Mark D. Wilkinson, and Ewan Birney. The bioperl toolkit: Perl modules for the life sciences. *Genome research*, 12(10):1611–1618, Oct 2002. 12368254[pmid].

[118] Martin Steinegger and Johannes Söding. Mmseqs2 enables sensitive protein sequence searching for the analysis of massive data sets. *Nature Biotechnology*, 35(11):1026–1028, 2017.

[119] Zachary D. Stephens, Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, Miles J. Efron, Ravishankar Iyer, Michael C. Schatz, Saurabh Sinha, and Gene E. Robinson. Big Data: Astronomical or Genomical? *PLoS biology*, 13(7):e1002195, July 2015.

[120] Hajime Suzuki and Masahiro Kasahara. Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC Bioinformatics*, 19(1):45, Feb 2018.

[121] Rust Team. The MIR, 2013.

[122] M. Vasimuddin, S. Misra, H. Li, and S. Aluru. Efficient architecture-aware acceleration of bwa-mem for multicore systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 314–324, May 2019.

[123] K Voss, J Gentry, and G Van der Auwera. Full-stack genomics pipelining with gatk4 +wdl +cromwell. In *18th Annual Bioinformatics Open Source Conference*, page poster, 2017.

[124] Martin Šošić and Mile Šikić. Edlib: a C/C++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9):1394–1395, 01 2017.

[125] Rajan Walia, Chung chieh Shan, and Sam Tobin-Hochstadt. Sham: A DSL for Fast DSLs, 2020.

[126] Ben J. Ward. *BioJulia*, accessed November 19, 2020. `https://biojulia.net`.

[127] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, December 1994.

[128] Deniz Yorukoglu, Yun William Yu, Jian Peng, and Bonnie Berger. Compressive mapping for next-generation sequencing. *Nat Biotech*, 34(4):374–376, 2016. Opinion and Comment.

[129] Matei Zaharia, William J. Bolosky, Kristal Curtis, Armando Fox, David A. Patterson, Scott Shenker, Ion Stoica, Richard M. Karp, and Taylor Sittler. Faster and more accurate sequence alignment with SNAP. *CoRR*, abs/1111.5572, 2011.

[130] Jing Zhang, Heshan Lin, Pavan Balaji, and Wu-chun Feng. Optimizing burrows-wheeler transform-based sequence alignment on multicore architectures. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 377–384. IEEE, 2013.

[131] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA):121:1–121:30, October 2018.

[132] Grace XY Zheng, Billy T. Lau, Michael Schnall-Levin, Mirna Jarosz, John M. Bell, Christopher M. Hindson, Sofia Kyriazopoulou-Panagiotopoulou, Donald A. Masquelier, Landon Merrill, Jessica M. Terry, Patrice A. Mudivarti, Paul W. Wyatt, Rajiv Bharadwaj, Anthony J. Makarewicz, Yuan Li, Phillip Belgrader, Andrew D. Price, Adam J. Lowe, Patrick Marks, Gerard M. Vurens, Paul Hardenbol, Luz Montesclaros, Melissa Luo, Lawrence Greenfield, Alexander Wong, David E. Birch, Steven W. Short, Keith P. Bjornson, Pranav Patel, Erik S. Hopmans, Christina Wood, Sukhvinder Kaur, Glenn K. Lockwood, David Stafford, Joshua P. Delaney, Indira Wu, Heather S. Ordonez, Susan M. Grimes, Stephanie Greer, Josephine Y. Lee, Kamila Belhocine, Kristina M. Giorda, William H. Heaton, Geoffrey P. McDermott, Zachary W. Bent, Francesca Meschi, Nikola O. Kondov, Ryan Wilson, Jorge A. Bernate, Shawn Gauby, Alex Kindwall, Clara Bermejo, Adrian N. Fehr, Adrian Chan, Serge Saxonov, Kevin D. Ness, Benjamin J. Hindson, and Hanlee P. Ji. Haplotyping germline and cancer genomes using high-throughput linked-read sequencing. *Nat Biotechnol*, 34(3):303–311, Mar 2016. 26829319[pmid].