# Codon: A Framework for Pythonic Domain-Specific Languages

by

Gabriel L. Ramirez

S.B. Computer Science and Engineering, Massachusetts Institute of Technology, 2021

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

© Massachusetts Institute of Technology 2021. All rights reserved.

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
May 14, 2021

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Saman P. Amarasinghe
Professor
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

# Codon: A Framework for Pythonic Domain-Specific Languages

by

Gabriel L. Ramirez

Submitted to the Department of Electrical Engineering and Computer Science
on May 14, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

Static languages like C++ provide deep compiler support for optimization and analysis, enabling high performance at the cost of burdening users with a low-level interface. DSLs, or domain-specific languages, have traditionally complemented static languages by adding custom idioms and optimizations in their particular areas. Unfortunately, however, static languages are increasingly sidelined in many domains by dynamic languages such as Python and Ruby, which, despite their flexibility, suffer from lower performance. In this thesis, we propose Codon, a framework for implementing high-performance DSLs based on Python. Whereas previously achieving high-performance in dynamic languages has been difficult, often requiring separate compiled libraries, we provide a robust base for analysis and optimization on a purely Pythonic base, bridging the gap between dynamic and static languages. By combining a purpose-built type checker and novel intermediate representation, Codon enables developers to create intrinsically performant, modular DSLs with minimal implementation effort. We validate this approach by showcasing several Codon DSLs, all of which achieve sizeable speed-ups over Python and sometimes even C++. We further show that Codon can be easily used to implement a variety of analyses and passes. Thus, Codon enables a new class of DSLs that maintain dynamism and expressiveness, without compromising performance.

Thesis Supervisor: Saman P. Amarasinghe
Title: Professor

# Acknowledgments

I would first like to thank my advisor of 4 years, Saman Amarasinghe. Not long ago, I sat in his office and asked for help finding a systems UROP. Saman suggested working with his COMMIT group, and, despite never touching a compiler before, accepted. Interestingly enough, everything I've worked on since has been related to them, but I guess, as Saman says, "once you find a hammer, everything is a nail."

Special thanks also go to my collaborators Ariya Shajii and Ibrahim Numanagić: this project has been a truly collaborative work that would have been impossible without their mentorship and contributions. I would also like to acknowledge the unending patience of Jessica Ray and Ajay Brahmakshatriya, Codon's first users. Apologies for all the bugs and thank you for your helpful comments.

Without a doubt, completing this project would have been impossible without the support of my friends Nyle, Sam, and Wilson, but especially my partner Katie. I'm sure it is not pleasant to listen to me ramble about intermediate representations for hours at a time, but they handled it with grace and offered me invaluable feedback in return.

Finally, I would like express my deepest appreciation for my family, parents — Irene and Bobby — and my brother — Pablo. Most importantly, I would like to dedicate this thesis to my beloved Abuelita Querida. Her strength and perseverance was and still is a constant inspiration.

# Previous Publications of This Work

This thesis' results are primarily derived from our collaborative submission to OOPSLA 2021 [32], co-authored with Ariya Shajii, Jessica Ray, Haris Smaljović, Bonnie Berger, Ibrahim Numanagić, and Saman Amarsinghe. Additionally, the idea for statically typing Python, as well as the framework's base, derive from the original Seq language [37]. Throughout, I use figures from the OOPSLA submission, marking them with a citation, and adapt text, as shown below:

- Chapter 4, specifically the type checker section, is adapted from the equivalent section in the publication. In the original paper, the type checker section was initially drafted by Ibrahim Numanagić and edited primarily by Ariya Shajii and myself.
- Chapter 5 is adapted from the equivalent section in the publication. With the exception of the code generation portion, which was drafted by Ariya Shajii and is reproduced in its entirety, the original section was drafted by myself and edited collaboratively.
- Chapter 6's results are adapted from the publication's results section, which was collaboratively drafted and edited.
- Chapter 9 extends the equivalent section in the publication, which was originally drafted by myself and collaboratively edited.
- Appendix B is partially replicated from the equivalent appendix in the publication, which was originally drafted by Ariya Shajii and collaboratively edited.

As far as the implementation of the paper, I was primarily responsible for the conceptualization, design, and implementation of the intermediate representation. Furthermore, I was responsible for implementing standard analyses/passes as well as Pythonic optimizations. For context and evaluation, this thesis discusses work performed primarily by collaborators. Special credit goes to Ariya Shajii for the implementation of Codon's code generation and Ibrahim Numanagić for implementing

the framework's frontend. Both are responsible for Codon's extensive standard library. Finally, collaborators were responsible for the implementations of the Codon DSLs.

# Contents


**1 Introduction** **17**

1.1 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

1.2 Thesis Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

**2 Background** **21**

2.1 Type Systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

2.1.1 Varieties of Typed Languages . . . . . . . . . . . . . . . . . . 21

2.1.2 Python's Type System . . . . . . . . . . . . . . . . . . . . . 23

2.2 Compilation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

2.2.1 Stages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

2.2.2 Types of IRs . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

**3 Codon Primer** **27**

3.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

3.2 Compilation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

3.3 Type System . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

3.4 Syntax & Semantics . . . . . . . . . . . . . . . . . . . . . . . . . . 30

**4 Language Frontend** **33**

4.1 Parsing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

4.1.1 AST Simplification . . . . . . . . . . . . . . . . . . . . . . . 33

4.2 Type Checking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

4.2.1 Hindley-Milner-Damas Type Inference . . . . . . . . . . . . . 35

4.2.2 Localized Type System with Delayed Instantiation . . . . . . 36

**5 Intermediate Representation 43**

5.1 Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

5.1.1 Converting to CIR . . . . . . . . . . . . . . . . . . . . . . 45

5.1.2 A Basic Example . . . . . . . . . . . . . . . . . . . . . . . 45

5.1.3 Bidirectionality . . . . . . . . . . . . . . . . . . . . . . . . 45

5.1.4 Extensibility . . . . . . . . . . . . . . . . . . . . . . . . . . 47

5.1.5 Code Generation . . . . . . . . . . . . . . . . . . . . . . . 49

5.2 Passes & Analyses . . . . . . . . . . . . . . . . . . . . . . . . 51

5.3 Previous Design . . . . . . . . . . . . . . . . . . . . . . . . . 51

**6 Creating DSLs with Codon 55**

6.1 Python . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

6.1.1 Standard Codon . . . . . . . . . . . . . . . . . . . . . . . 55

6.1.2 Dictionary Access . . . . . . . . . . . . . . . . . . . . . . . 56

6.1.3 String Manipulation & I/O . . . . . . . . . . . . . . . . . . 59

6.2 Seq . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

6.2.1 Custom Nodes . . . . . . . . . . . . . . . . . . . . . . . . . 60

6.2.2 Prefetch Optimization . . . . . . . . . . . . . . . . . . . . 60

6.2.3 Inter-align Optimization . . . . . . . . . . . . . . . . . . . 61

6.3 Sequre . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

6.4 CoLa . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

**7 Standard Analyses & Passes 67**

7.1 Analyses . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

7.1.1 Control-flow Graphs . . . . . . . . . . . . . . . . . . . . . 67

7.1.2 Reaching Definitions . . . . . . . . . . . . . . . . . . . . . 68

7.1.3 Dominators . . . . . . . . . . . . . . . . . . . . . . . . . . 69

7.2 Passes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

7.2.1 Loop Lowering . . . . . . . . . . . . . . . . . . . . . . . . 69

7.2.2 Code Simplification . . . 70

**8 Related Works** **73**

8.1 Augmenting Dynamic Languages . . . 73

8.1.1 Type Checking . . . 73

8.1.2 Performance Optimization . . . 74

8.2 DSL Frameworks . . . 74

8.3 Intermediate Representations . . . 75

**9 Conclusion** **77**

9.1 Future Work . . . 77

**A Supplemental Listings** **79**

A.1 AST Nodes . . . 79

A.2 CIR Nodes . . . 82

A.3 Legacy CIR Nodes . . . 84

**B CIR Utilities** **85**

B.1 IR Manipulation . . . 85

B.2 Miscellaneous Tools . . . 87

**C Benchmark Information** **89**

C.1 Pythonic Benchmarks . . . 89

C.1.1 PyPerformance . . . 89

C.1.2 Word Count . . . 90

C.2 Seq Benchmarks . . . 91

C.2.1 Prefetch . . . 91

# List of Figures


2-1 Classic compilation pipeline. . . . . . . . . . . . . . . . . . . . . . . . 24

3-1 A short example of Codon source. . . . . . . . . . . . . . . . . . . . . 27
3-2 Codon's compilation pipeline. . . . . . . . . . . . . . . . . . . . . . . . 28
3-3 Generics and type inference in Codon. . . . . . . . . . . . . . . . . . 30

4-1 Hello World AST. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4-2 Invalid Codon when using a unidirectional type checker. . . . . . . . 35
4-3 Example of type inference and function instantiation. . . . . . . . . . 37
4-4 Static evaluation of `if` statements. . . . . . . . . . . . . . . . . . . . 38
4-5 Overloading methods. . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

5-1 CIR class hierarchy. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
5-2 LLVM implementation of `int.__add__`. . . . . . . . . . . . . . . . . . . 45
5-3 CIR equivalent of Fibonacci function. . . . . . . . . . . . . . . . . . . 46
5-4 C++ implementation of a multiply-by-power-of-2 transformation. . . . 48
5-5 Implementation of 32-bit float type. . . . . . . . . . . . . . . . . . . . 50
5-6 Implementation of a constant folding pass. . . . . . . . . . . . . . . . 52
5-7 Legacy CIR structure. . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

6-1 Performance comparison on various Pythonic benchmarks. . . . . . . 55
6-2 Codon component of dictionary optimization pass. . . . . . . . . . . . 56
6-3 Simplified C++ component of dictionary optimization pass. . . . . . . 57
6-4 Codon implementation of word-counting benchmark. . . . . . . . . . 58
6-5 Speedup for various word-count implementations. . . . . . . . . . . . 58

6-6 Simplified C++ implementation of the string addition folder. . . . . . 59
6-7 Codon implementation of the dynamic scheduler. . . . . . . . . . . . 61
6-8 Simplified C++ implementation of the prefetch function transformation. 62
6-9 Runtimes for FM queries at various $k$. . . . . . . . . . . . . . . . . . 63

7-1 Example `for` loop control-flow graph. . . . . . . . . . . . . . . . . . . 68
7-2 Simplified CIR equivalent of a `for` loop. . . . . . . . . . . . . . . . . 69
7-3 Simplified C++ implementation of `for` loop lowering. . . . . . . . . . 70
7-4 Simplified C++ implementation of constant propagation. . . . . . . . . 72

C-1 C++ implementation of word-counting benchmark. . . . . . . . . . . . 90
C-2 Seq prefetching benchmark. . . . . . . . . . . . . . . . . . . . . . . . . 91

# List of Tables

3.1 Codon types. . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

4.1 Optional value unwrapping/wrapping rules. . . . . . . . . . . . . . . 39

5.1 Listing of CIR node varieties. . . . . . . . . . . . . . . . . . . . . . 44
5.2 Listing of CIR customizable nodes. . . . . . . . . . . . . . . . . . . 49

A.1 Listing of simple AST statements. . . . . . . . . . . . . . . . . . . . 79
A.2 Listing of complex AST statements. . . . . . . . . . . . . . . . . . . 80
A.3 Listing of AST expressions. . . . . . . . . . . . . . . . . . . . . . . 81
A.4 Listing of builtin CIR types. . . . . . . . . . . . . . . . . . . . . . . 82
A.5 Listing of CIR variables. . . . . . . . . . . . . . . . . . . . . . . . . 82
A.6 Listing of CIR values. . . . . . . . . . . . . . . . . . . . . . . . . . 83
A.7 Listing of legacy CIR instructions, l-values, r-values, operands, and terminators. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84

B.1 Listing of select `Module` utilities. . . . . . . . . . . . . . . . . . . . 85
B.2 Listing of select `Node` utilities. . . . . . . . . . . . . . . . . . . . . 85
B.3 Listing of select `Value` utilities. . . . . . . . . . . . . . . . . . . . . 86
B.4 Listing of miscellaneous utilities. . . . . . . . . . . . . . . . . . . . . 87

# Chapter 1

# Introduction

Traditionally, achieving high performance entailed developing in a statically-typed, compiled language like C or C++. Such languages provide sophisticated compilers which extensively analyze and optimize their code, at the expense of providing users with a highly low-level interface. For many tasks, however, this lower-level view is inconvenient and results in verbose, un-maintainable code. DSLs, or domain specific languages, arose to increase the expressiveness of lower-level languages by extending them with custom idioms for specific use-cases. These languages, typically embedded in a high-performance base language, inherit the compiler support of their parents, all the while leveraging increased semantic information to achieve even better performance.

However, the lower-level languages used to create traditional DSLs like Halide [31] are falling out of favor to dynamic languages like Python and Ruby, threatening DSL adoption. Dynamic languages provide a considerably more flexible interface for developers, who benefit from increased productivity and easier prototyping. Unfortunately, however, these languages' lax rules make them notoriously difficult to analyze and optimize: for Python, many tools such as PyPy [35] opt to substitute a more performant interpreter for the standard implementation. Still others such as Cython [4] retrofit type-checking and compilation for specific subsets of Python. Unfortunately, these tools are unable to optimize large swaths of Pythonic code, minimizing their advantages. Another approach aims to create the illusion of high-performance though

externally implemented libraries like NumPy [14] and TensorFlow [1]; nonetheless, these libraries are essentially auxiliary to the core language and are unable to cover all cases [32]. Consequently, general-purpose dynamic code often runs orders of magnitude slower than in compiled languages.

The Seq language [37], a DSL for bioinformatics and genetics, addresses the popularity of dynamic languages by borrowing Python's syntax, semantics, and memory model. Rather than depending on its high-overhead interpreter, however, Seq simulates Python's dynamism at compile-time by substituting a thoroughly compatible and statically-typing compiler, composing it with highly-optimized domain-specific features. While the language achieves "C-like performance," it lacks compatibility with a few common Python idioms like empty dictionary literals (i.e. `x = {}`), its type-checking system lacking full support for *duck-typing*. Further, the language is intimately tied to bioinformatics and does not easily support extension.

## 1.1 Contributions

In this thesis, we generalize the language's compiler into Codon, an end-to-end framework for creating custom Pythonic DSLs. To enable better compatibility with Python, we augment Seq's base with a Hindley-Milner-Dumas type checker. Additionally, we introduce a novel intermediate representation (IR) that enables easy optimization and extensibility. Finally, we incorporate a rich set of utilities and standard optimizations. Taken together, these extensions enable the framework to contribute the following[1]:

- **Comprehensive type-checking of dynamic source.** We show using a technique called *monomorphization*, typically applied to strongly-typed languages like C++, allows Codon to achieve impressive coverage of Pythonic syntax.

- **Bidirectional intermediate representation.** Unlike in traditional compilers, Codon's IR passes can re-invoke the type checker to generate new nodes and types. This allows for considerable flexibility and generality in pass writing,

[1] These contributions are adapted from those of [32].

allowing most implementation effort to be offloaded to the source itself.

- **Framework for High-Performance Pythonic DSLs.** Codon enables DSLs with Pythonic syntax to be rapidly deployed; these languages inherit Codon's robust type system and optimization support. Most importantly, they are intrinsically high-performance.

Codon provides a novel solution for implementing easily-adoptable DSLs with minimal development effort. Despite their flexible and familiar syntax, Codon DSLs are intrinsically high-performance, inheriting the framework's highly-efficient implementation. Thus, Codon introduces a new class of DSLs, which, by adopting zero-overhead dynamic syntax, do not compromise between flexibility and performance.

## 1.2 Thesis Structure

In Chapter 2, we briefly explore the internals of compilation, specifically type-checking and intermediate representations; Chapter 3 applies these concepts to introduce the Codon framework. Chapter 4 explores Codon's frontend and type-checker in depth. Chapter 5 outlines the features and operation of Codon's novel IR. In Chapter 6, we apply Codon to various optimizations and DSLs. In Chapter 7, we overview some of Codon's standard passes and analyses. Finally, in Chapter 8, we address related works and conclude with Chapter 9.

# Chapter 2

# Background

In this chapter, we explore relevant topics on compilation and type systems, paying special attention to concepts used by Codon's compiler.

## 2.1 Type Systems

In assembly and machine code, all data is represented and manipulated in the same way — as an ordered list of bits separated into *words*. Consider data of the form `struct X {uint16_t x; uint16_t y}`; accessing either the $x$ or $y$ members requires careful address calculation and/or bitwise manipulation. Beyond being difficult to work with, this approach is extremely error prone: users can easily conflate data organized differently (i.e. `X` with reversed members). Consequently, most higher level programming languages have adopted notions of *typing*, which enforce some degree of separation between different varieties of data. At minimum, type checking verifies that data accesses and function calls are valid, or *sound*.

### 2.1.1 Varieties of Typed Languages

Within the space of typed languages, some systems perform type soundness checks at compile time while others defer the checks until the program is run, the former referred to as static typing and the latter as dynamic typing [29]. Dynamic languages

tend to be less verbose and easier to use: freed from having to constantly consider types, developers can more rapidly prototype. Further, dynamic languages can offer more opportunities for flexibility; for example, variables can have differing type based on the particular execution path. Unfortunately, however, deferring all type checking to runtime has vast performance impacts since nearly every operation must check for type correctness. Further, debugging can often be more difficult in dynamic languages since type checks may occur long after the code was written and first run [32]. Conversely, static languages can be difficult to master, lacking support for the same expressiveness as dynamic languages, but offer superior performance.

Another divide results from the *strictness* of type systems: some languages like C and C++ allow unsafe conversion between types, whereas others like Java allow this only in certain situations with runtime checking. For example, a weakly typed language like C may allow users to coerce pointers to one type to another pointer type without complaint. Related to these concepts is *duck typing*, which extends dynamic typing to rely on the so-called duck test — “if it quacks like a duck, it’s a duck” [27]. Duck-typed languages rely on interface checks to verify soundness at runtime. For example, a duck-typed function can accept a parameter `x` and access various attributes and functions; so long as these accesses are valid at runtime, the function is deemed sound. Consequently, languages that employ duck typing restrict runtime checks to determining the existence of requested methods or members rather than enforcing strict type checking.

Given the advantages of dynamic systems, much effort has been invested in allowing statically checked languages to achieve dynamic qualities. Rather than requiring manual annotation, many statically checked languages can infer the types of variables, reducing developer overhead and verbosity. Similarly, statically typed languages can achieve an approximation of duck-typing through *parametric polymorphism*, which enables users to write functions that do not depend on the particular types of their inputs [29]. Despite these techniques, achieving full coverage of dynamic features is extremely challenging and often impossible, particularly when trying to apply a static system to a dynamic language.

For example, consider a dynamic dictionary type which can contain keys and values of any type. In a duck-typed language, code can operate over such dictionaries without added constraints, provided that each individual access and method call is legal. In statically-typed languages, however, the type of each object must be uniquely known, making this case difficult to handle. A similar issue arises from Python's ability to execute strings and manipulate functions at runtime — ahead-of-time compilation becomes impossible.

### 2.1.2 Python's Type System

As with all dynamically checked languages, Python defers type safety checks to runtime. Since the language is duck-typed, users are free to pass objects of any type into functions provided they implement the required interface. Further, the language provides runtime support for inspecting the types of variables, allowing logic and even return types to vary based on inputs. Nonetheless, Python is strongly typed, only allowing types to be converted based on strict rules.

Given these characteristics, all types in Python are implemented as objects (or structured data with functions), albeit with some optimization for "primitive" types like `int` or `float`. Consequently, all operations, even simple addition and subtraction, are implemented as function calls.

## 2.2 Compilation

Beyond the different type systems, programming languages split into two categories: *interpreted* and *compiled*. Interpreted languages like Python rely on auxiliary programs to execute code and provide facilities like memory management and garbage collection. Conversely, compiled languages are translated into machine code before runtime and typically have better performance. Additionally, some languages adopt aspects of both of these categories through JIT — just-in-time compilation. JIT, used notably by Java, allows code to be partially interpreted, with hot-spots being compiled before runtime. This thesis specifically focuses on wholly-compiled languages.

### 2.2.1 Stages

$$Parsing \xrightarrow{AST} Typing \xrightarrow{AST_t} Optimization \xrightarrow{AST_t} Codegen \xrightarrow{Bytecode} Execution$$

Figure 2-1: Classic compilation pipeline.

Classically, compilation proceeds as outlined in Figure 2-1: source code is directly translated by a parser into an abstract syntax tree, which maintains a 1-1 correspondence with the underlying source. In statically typed languages, this AST is then checked for soundness before being optimized and finally converted into the target source. This approach has the benefit of being simple to implement, particularly if the compiler targets some other language like LLVM IR [19] rather than machine code. Unfortunately, ASTs, by virtue of their close correspondence with the source, contain the full complexity of the language. This makes performing complex manipulations or analyses extremely tedious, as every optimization must consider every possible variation of its target pattern. As such, many compilers insert simplified intermediate representations (IRs) between the typing and optimization stages, making pass-writing easier.

### 2.2.2 Types of IRs

Typically, intermediate representations are hierarchical or linear [2]. IRs with tree-like structure tend to at least superficially resemble the underlying source; indeed, ASTs are one example of hierarchical IRs. Explicitly representing the source's structure can be advantageous for highly language-dependent optimizations that benefit from increased semantic information. In contrast, linear IRs such as LLVM IR [19] deconstruct the source into a flat structure resembling assembly — sections of code without branches are organized into *basic blocks* linked together with *terminators*. This results in linear IRs being considerably lowered from the original source, a key benefit for performing certain types of analyses.

Orthogonally, IRs also differ in how they represent variables. In source code, memory locations can be written to and read an arbitrary number of times. While some

IRs maintain these semantics, others transform the code into single-static assignment (SSA) form [33], in which variables are assigned to exactly once. As in linear IRs, this form makes several analyses more straightforward.

# Chapter 3

# Codon Primer

In this chapter, we provide an overview of Codon's compilation pipeline, syntax, and semantics.

## 3.1 Overview

```
def fib(n):
  if n < 2:
    return 1
  else:
    return fib(n - 1) + fib(n - 2)
```

Figure 3-1: A short example of Codon source.

Figure 3-1 contains a valid Codon Fibonacci function (originally shown in [32]). In particular, note that the framework's syntax and semantics are exactly identical to those of Python. Indeed, this example is easily run using a standard Python interpreter, giving the same results.

As a successor to the Seq [37] language, Codon mirrors its approach to maintaining compatibility with Python. Wherever possible, Codon utilizes the same keywords and syntax as Python. Additionally, Codon maintains the same memory model and often requires no explicit type annotations. To that end, many Python programs can be run without further intervention.

While Codon is able to achieve impressive coverage of the vast majority of Python, below are some key examples of features not yet supported:

- **Inheritance:** Codon does not currently support polymorphism in the traditional sense. By providing an "extension" idiom, users can add methods to preexisting types.
- **Monotype:** Each variable in Codon must have exactly one type.
- **Mixing types:** Unlike in Python, users are not allowed to mix types in collections. For example, Python lists can contain objects of arbitrary types, but in Codon all the elements must be the same type.
- **Runtime patching:** All methods and functions are realized at compile time. As such, Codon cannot support dynamically replacing functions.

## 3.2 Compilation

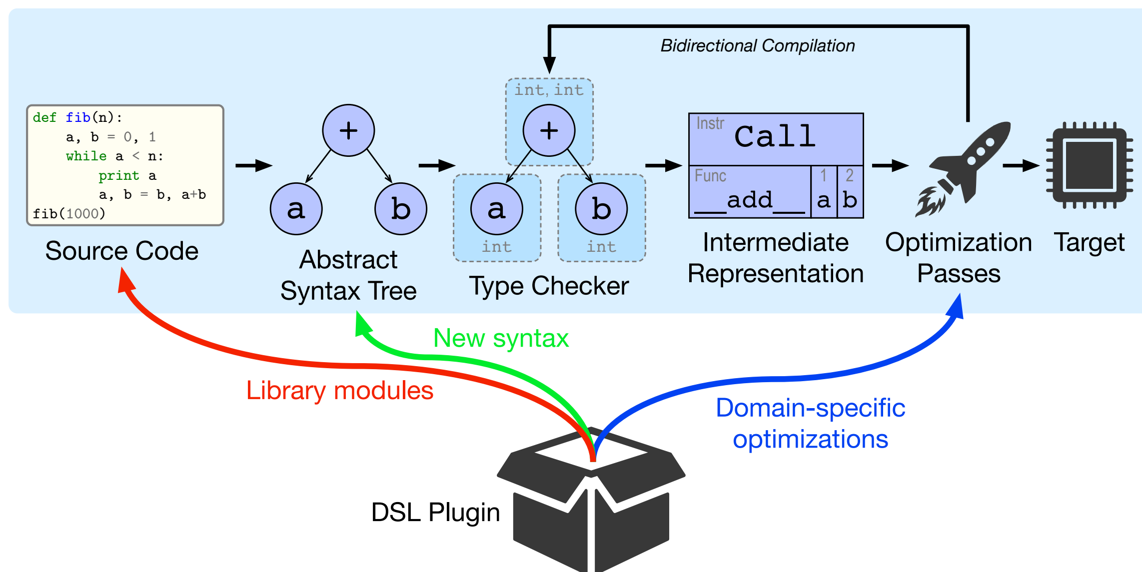


Figure 3-2: Codon's compilation pipeline.

Codon's pipeline, shown in Figure 3-2 [32], begins with an OCaml parser that reads the source code into an AST, which is federated into the main C++ compiler

and simplified. We then apply a modified Hindley-Milner-Damas type checker to infer unspecified types and check for soundness. This results in a typed AST that we then convert into an intermediate representation (CIR, or Codon IR). The resultant CIR code is then optimized and converted into LLVM IR, as in the original Seq [37] implementation. Since we make use of LLVM as a backend, we are able to take advantage of its rich optimization library and target multiple architectures without additional work.

DSL developers are able to customize nearly all aspects of this pipeline. In particular, plugins can bundle standard library code, as Seq does with its `bio` standard library. These libraries can be augmented with domain-specific syntax and optimizations. Importantly, these customizations are auxiliary to the core Codon pipeline, enabling easy composition. In fact, we expect that users could use multiple DSL components in a single program.

## 3.3 Type System

| Type | Examples | Python? | Codon Representation |
|---|---|---|---|
| Function | `Function[void, [int]]` | ✓ | LLVM function pointer. |
| Primitive | `int`, `float` | ✓ | LLVM primitive. |
| Reference | `Dict`, `Set` | ✓ | Pointer to an LLVM struct. |
| Record | `Tuple` | ✗ | LLVM struct |

Table 3.1: Codon types.

Codon maintains the original Seq [37] implementation's general static type system, which is briefly listed in Table 3.1. Unlike in Python, types are mapped to the simplest possible underlying data types. As such, primitives are represented with lightweight LLVM types, but are nonetheless first-class members of Codon's object-oriented structure. Compound types such as classes are represented as "LLVM aggregate type[s] or pointer[s] to one" [37]. In particular, note that Codon makes a distinction between record and reference types, one that does not exist in Python.

Record types are essentially passed by value (i.e. like C structs) and reference types are passed by pointer.

As in C++ and other languages like Java, Codon supports generic types and functions. To enable compatibility with Python's duck-typing paradigm, functions and types without specific type annotations are considered generic. During type checking, unknown types are inferred and types/functions are instantiated using the appropriate generic parameters. The details of this checking and instantiating process are outlined in Chapter 4.

```
class Wrapper[T]:
    val: T
    def __init__(self, v: T):
        self.val = v
    def output(self):
        print f'wrapped({str(self.val)})'

x = Wrapper(1)
x.out()
```

Figure 3-3: Generics and type inference in Codon.

Figure 3-3 shows a basic example of generics and automatic type realization. We define a basic class `Wrapper[T]` that is generic on parameter `T`. This generic parameter is used to define a variable `val`, which is set in `__init__` and used in `output`. In implementing the class, we make use of Codon's support for explicit type annotations to denote that the type of generic `T` is identical to that of `val` and the initialization parameter. Once this initial legwork is complete, however, users are able to use the class without explicitly specifying the type of `T`; in the example, `T` is inferred to be `int`. Indeed, Codon's type checker is able to infer the types of generics in most cases, preventing users from having to manually annotate their code.

## 3.4 Syntax & Semantics

As can be seen in the examples above, Codon borrows Python's keyword and general structure. In addition to differences due to the type checker, Codon enforces a few

conventions not present in Python. In particular, Codon uses strict scoping rules — variables are only usable in their specific scope — unlike Python. Additionally, Codon adds a few language features not present in Python, including pattern matching, pipelines, and partial function features.

Codon's support pattern matching using a `match` statement conceptually similar to a large `if` statement tree. This construct supports matching on strings and entire objects. Pipelines, denoted with the `|>` sigil, enable developers to compose standard functions and generators in a succinct manner. We additionally support parallel pipelines using Tapir LLVM [36]. Finally, we allow users to manipulate partial applications of functions.

# Chapter 4

# Language Frontend

This chapter explores in depth the frontend of Codon's compiler. We first examine Codon's parser and simplification passes, before proceeding into a discussion of the type checker.

## 4.1 Parsing

As overviewed in Chapter 2, parsing is the first step of compilation, by which source code is ingested and converted into an in-memory representation. Typically, languages will make use of either hand-written tools or parser generators that employ a given formal grammar. Codon and Seq [37] both employ an OCaml-based parser generator called Menhir[1] [30]. The resulting OCaml AST is federated into C++ and manipulated via a few simplification passes, outlined below.

### 4.1.1 AST Simplification

Codon's AST is composed of expressions, which combine other expressions to produce new values, and statements, which are typically structural in nature. For example, *for* loops are represented as statements that contain the iteration value, itself an expression. The statements and expressions listed in Tables A.1, A.2, and A.3 (all in Appendix A) are sufficient to represent all elements of Codon source.

[1] This portion of the frontend will be replaced in a future revision.

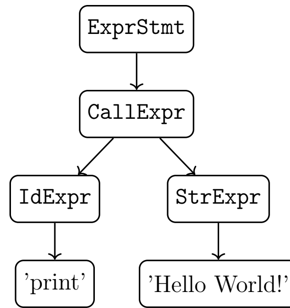


Figure 4-1: Hello World AST.

Notably, however, the full complexity of this structure is not needed to accurately represent computation in Codon. In particular, the framework mimics Python's use of *magic functions*, allowing the parser to simplify the AST by mechanically replacing certain nodes with their function equivalents. We consider a few other important transformations below.

- **Literals:** `IntExpr`, `FloatExpr`, and `StrExpr` can contain prefixes and suffixes. If these are present, the constants are replaced with function calls. For example, this mechanism allows binary constants like `0b110`.

- **Container Literals:** Literals and comprehensions are transformed into a call of the appropriate container initialization functions and `__setitem__` calls.

- **Classes:** The simplifier adds various implicit magic methods to `ClassStmt` and flattens them into a type and various functions.

- **Pattern matching:** `match` statements are fully resolved into `if` statement trees with appropriate function calls at this stage.

In Figure 4-1, we show the equivalent AST for a simple `print 'Hello World'` statement. As per the rules shown in Tables A.1, A.2, and A.3, the `print` statement is desugared to a call and the constant is replicated as is.

## 4.2 Type Checking

Unlike the classic Python implementation, Codon uses ahead-of-time compilation to avoid relying on a virtual machine. As such, the compiler must determine the precise data type and memory location of every value in the program; to avoid requiring type annotations, this requires an inference algorithm. Codon's predecessor [37] utilized a simple uni-directional type checker akin to C++'s; analogous checkers for Ruby [23] and Python [21] take the same approach.

Uni-directional type checkers simply iterate through the nodes of a program, progressively deciding types. If a type cannot be decided at its occurrence, an error is thrown; thus, any ambiguity must be resolved with an explicit annotation. Unfortunately, Pythonic code contains bountiful ambiguity, particularly around container types, making this approach inappropriate. Furthermore, it does not allow easy implementation of lambda functions or several other important constructs.

```
x = [] # illegal, x's type is unknown
x.append(1)

f = lambda y: y + 1 # illegal, f's type is unknown
```

Figure 4-2: Invalid Codon when using a unidirectional type checker.

Figure 4-2 contains an example of Codon source that would be unusable with a undirectional type checker: the types of both `x` and `f` could not be determined. We address these shortcomings by proposing a new type system, *localized type system with delayed instantiation (LTS-DI)*, which extends the classical Hindley-Milner-Damas bidirectional type inference algorithm for easier use in Pythonic code [29]. To that end, we make several departures from the "canonical ML" rules to enable better compatibility without burdening developers.

### 4.2.1 Hindley-Milner-Damas Type Inference

Hindley-Milner-Damas [15, 25, 10] type checkers rely on the notions of *type constraints*, which designate rules for unbound types, and *unification*, which solves the

various constraints [29]. Intuitively, these checkers iteratively propagate type information from known types (i.e. constants) to determine the types of all relevant expressions in the program. Such checkers are bidirectional in that the type of each expression does not have to be known immediately. Instead, types that may not be realizable at first glance may become realizable when further constraints are known. This allows for considerably more flexibility than unidirectional checkers.

### 4.2.2 Localized Type System with Delayed Instantiation

We introduce LTS-DI, a novel bidirectional type system inspired by Hindley-Milner-Damas [15, 25, 10] type systems. LTS-DI, which supports parametric polymorphism, allows types to be either *concrete* (i.e. `int` and `List[str]`) or *generic* (i.e. parameterized). Importantly, LTS-DI does not require all types to be decided immediately: generics and indeed any type can be left ambiguous until resolved by later constraints.

Type checking begins with a context $\Gamma$ initialized with various basic types, such as primitives and builtin generics. Each type and function declaration, together with variable assignments, adds to this context. Further, as types become known, unification proceeds as in the traditional Hindley-Milner-Damas manner. Consider the simple program `a = None; a = Optional(1)`. When the checker encounters the first assignment, `Optional[`$\alpha$`]` is added to $\Gamma$; at this stage, inferring the concrete type of $\alpha$ is not possible. On the second assignment, however, the checker can unify `Optional[`$\alpha$`]` and `Optional[int]`, giving $\alpha =$ `int`.

This procedure proceeds for the entire program, until either all types have been realized or no more unifications are possible. Naturally, if checking ends before all types have been determined, further processing is impossible and an error is raised. Figure 4-3 [32] presents a brief example of this process.

LTS-DI's approach to checking functions is inspired by Standard ML-like [26, 29] type systems, which use a notion of *let-polymorphism* to implement functions. Since Python programs rely heavily on duck-typing, we depart from Standard ML's approach by delaying generic function type inference *until* the function is instantiated with bound types. This causes each unique combination of argument types to produce

```
def List_append(self: List[T], what: T):
  ...

a = Optional()       # a: Optional[α]
b = List()           # b: List[β]
for i in range(3):   # i: int
  List_append(b, i)  # List_append:
                     #  Function[[List[γ], γ], η]
  # Type inference does the following steps:
  # unify:     γ ← β ←int
  # realize:  List_append → List_append_int
  # unify:     η ←void
a = Optional(b)      # unify: α ←List[int]
```

Figure 4-3: Example of type inference and function instantiation.

a completely new function (or instantiation), each of which is typed separately.

This technique, called *monomorphization*, is used by many static languages like C++, specifically in its templating system. In LTS-DI, each instantiation's body and return type are inferred using only the argument types and generics supplied at the time of realization. Intrinsically, this prevents function calls from being processed until all argument types are known; the system simply defers such calls until more constraints on the arguments become apparent. While this approach results in moderately more total instantiations, it, together with parametric polymorphism, allows LTS-DI to faithfully simulate Python's runtime duck typing ahead of time. Importantly, equivalent function instantiations are collapsed by the LLVM backend, reducing the overhead of this procedure.

Unlike in ML-like systems, LTS-DI enforces *localization*: each function block is type checked separately in its own distinct typing context $\Gamma_f$. Consequently, each block within the function must be independently resolvable without information from other functions' blocks.

These two departures allow LTS-DI much greater flexibility with generic functions. In other ML-like systems, each function's type is determined by its first application. Consider the simple Pythonic snippet `print(1); print('hi')`. With a conventional type checker, the second expression would be illegal as the first application would

cause the checker to irrevocably infer that `print` takes an integer argument. LTS-DI resolves this issue by preserving functions' genericity after their first application. This strategy is maintained even through function passing: passed functions can in fact be instantiated differently depending on the supplied arguments to the overall function. Maintaining function genericity is instrumental in allowing LTS-DI to achieve a better level of compatibility with Python, specifically around its general lambda and decorator support.

#### 4.2.2.1 Static Evaluation

```
def foo(x):
    if isinstance(x, int):
        return 1
    else:
        x.append(1)

x: List[int] = []
foo(x) # legal, appends to x
print foo(1) # legal, prints 1
```

Figure 4-4: Static evaluation of `if` statements.

LTS-DI also differs from ML-like systems by providing support for static evaluation of `if`-statement conditions. In Python, conditioning branches and function operation on the type of an input is a common pattern. For example, Figure 4-4 shows a function that performs different actions based on whether its argument `x` is an integer. Most type checkers would treat these branches as runtime checks, over-eagerly rejecting the function as unsound. LTS-DI, however, treats certain branches as *compile-time* checks. To that end, the code is legal because only one of the branches is ever executed.

Many Codon methods, particularly `isinstance` and `hasattr`, can in fact be resolved at compile time. Codon borrows the concept of static expressions (akin to `constexpr`s in C++) and allows such expressions to be evaluated at compile time. If they appear in an `if`-statement's condition, the checker only evaluates branches if

| Pattern | Condition | Result |
|---|---|---|
| x = <v> | optional x | x = wrap(<v>) |
| f(<v>) | optional arg | f(wrap(<v>)) |
| f(<opt>) | non-optional arg | f(wrap(<opt>)) |
| <opt>.<member> | ✗ | unwrap(<opt>).<member> |
| <v> if <c> else <opt> | ✗ | wrap(<v>) if <c> else <opt> |
| <opt> if <c> else <v> | ✗ | <opt> if <c> else wrap(<v>) |

Table 4.1: Optional value unwrapping/wrapping rules.

they correspond to a true static value.

Codon's implementation of static evaluation also supports constant expressions. For example, the framework's variable length integer type is expressed as Int[N: int], where N is an integer generic. Constant expression generics behave in the same manner as standard ones, with different values resulting in totally different instantiations. Further, they can be combined with simple arithmetic operations to express complex constraints.

#### 4.2.2.2 Special Rules

**Optional Values** In Python, every value can be *None* or some value and is represented essentially a pointer. For performance reasons, Codon enforces a distinction between optional and non-optional values with its Optional[T] generic. While optional values have the added flexibility of supporting None, they must be *unwrapped* and checked for None before use. Similarly, passing a non-optional value to a function that expects an optional requires *wrapping*. Since this distinction does not exist in Python, we add compiler support for automatically wrapping/unwrapping optionals in the cases shown in Table 4.1; this obviates most manual optional manipulation.

**Functions/Methods** Codon allows users to create and manipulate partial functions either through Python's functools.partial construct or via a new ellipsis syntax (where f(42, ...) indicates a partial function call with only the first argument provided). Internally, we implement such partial calls as a unique named tuple

containing the provided arguments. As with vanilla functions, Codon allows partial functions to be generic and instantiated differently depending on future applications. To support lambdas and decorators, Codon automatically converts passed or returned functions into partial calls, allowing the framework to safely support complex decorators. Unfortunately, allowing generic function passing potentially results in different partial functions being incompatible, despite having identical argument types. In many cases, however, LTS-DI can automatically convert between compatible partial types.

```
class Calculator:
  val: int
  def __init__(self):
    self.val = 0

  def add(self, val):
    if isinstance(val, int):
      self.val += val
    elif isinstance(val, float):
      self.val += int(round(val))
    else:
      assert False

  def sub(self, val: int):
    self.val -= val

  def sub(self, val: float):
    self.val -= int(round(val))

x = Calculator()
x.add('1') # compiles, run-time error
x.sub('1') # fails, compile-time error
```

Figure 4-5: Overloading methods.

Codon additionally supports explicit function overloading through type annotation. While this can be supported with `isinstance` checks in Python, explicit overloading allows for cleaner, statically checked code. For example, in Figure 4-5, we show `add` and `sub` methods, implemented with `isinstance` and overloading respectively. While the `isinstance` method is more Pythonic, overloading ensures the

desired behavior occurs by rejecting the program if types do not match.

**Miscellaneous considerations** In Python, imports are performed at runtime and can be embedded in any location. Codon simulates this by wrapping import statements in functions that are called only by the first statement that reaches it. LTS-DI further automatically unwraps iterables and coerces integer and float types automatically. The checker als implements some support for traits (specifically `Callable` and `Generator`). Finally, Codon allows users to call Python through its `pyobj` interface; in many cases, the checker can automatically unwrap this object into a native Codon type.

**Limitations**

LTS-DI enables Codon to support the vast majority of Python, including comprehensions, iterators, generators, function manipulation, variable arguments, pattern matching, and decorators. Much Pythonic code is usable without any intervention. Additionally, the framework adds new features such as pipelines, static evaluation, and compile time typing.

Nonetheless, Codon lacks support for a few Pythonic features. Due to its compiled nature, the framework does not allow for dynamic modification of types (e.g. modifying dynamic method tables or adding or removing members at runtime). Additionally, LTS-DI does not currently support inheritance or dynamic polymorphism, but this support is planned for the next major release.

# Chapter 5

# Intermediate Representation

This chapter discusses Codon's intermediate representation, specifically its structure and features, before discussing previous iterations.

## 5.1 Design

Codon's IR (CIR) originates from the Seq [37] language's need for a flexible mid-layer on which to perform optimizations. This close dependence on the requirements of Seq have caused it to develop novel features that are ideal for a dynamic context, as well as a unique design.

CIR distills source code from the AST level to a vastly simplified representation. Taking inspiration from LLVM [19], the representation relies on the notion of *values* — computed results or instructions. Most values have fully-realized, non-generic types and, in keeping with single-static assignment (SSA) form, all can only be assigned once. To represent variables, CIR denotes them as memory locations, which can naturally be modified repeatedly. This hybrid SSA is convenient for both analyses and conversion to LLVM IR, the target of Codon compilation.

Unlike many IRs, CIR maintains a hierarchical structure, wherein values can be easily nested and combined. This hierarchy extends even to the realm of control-flow: rather than fully deconstruct the source into basic blocks, CIR maintains explicit nodes called *flows*. This approach is similar to that taken by other IRs such as

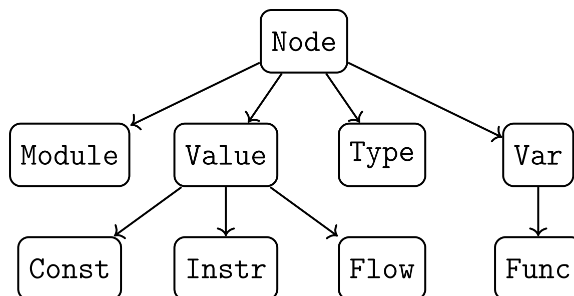

Figure 5-1: CIR class hierarchy.

Suif [40] and Taichi [16].

CIR's hierarchical structure is particularly essential due to the underlying language's dynamism. Since patterns such as ranged iteration and generators often obscure the underlying computation, maintaining the source's structure is crucial for optimization. One key example is the `for` flow: in Pythonic languages, the `for x in range(y)` pattern is exceedingly common; maintaining explicit loops allows Codon to easily recognize this pattern rather than having to decipher a maze of branches, as is done in lower-level IRs like LLVM IR.

In Figure 5-1, we show the class hierarchy of CIR nodes, providing more detail on each variety in Table 5.1. A full listing of CIR's nodes can be found in Tables A.4, A.5, and A.6, all in Appendix A.

| | **Description** | **LLVM Equivalent** | **Examples** |
|---|---|---|---|
| `Node` | Parent class. | N/a | All CIR nodes. |
| `Module` | Container for program. | `Module` | N/a |
| `Type` | Value type. | `Type` | `IntType`, `RecordType` |
| `Var` | Variable or function. | `AllocaInstr`, `GlobalVariable` | `Var`, `Func` |
| `Func` | Function. | `Function` | `BodiedFunc`, `LLVMFunc` |
| `Value` | Program value. | `Value` | `VarValue`, `Instr` |
| `Const` | Constant. | `Constant` | `IntConst`, `FloatConst` |
| `Instr` | Instruction. | `Constant` | `IntConst`, `FloatConst` |
| `Flow` | Control flow node. | Various | `ForFlow`, `IfFlow` |

Table 5.1: Listing of CIR node varieties.

### 5.1.1 Converting to CIR

The AST described in Chapter 4 can be easily transformed into CIR. Most statements are ported over directly into flows. For example, each `ForStmt` becomes a `ForFlow`. Expressions, with few exceptions, are translated into function calls, implemented using Pythonic magic methods. Wherever possible, these are implemented in Codon source.

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

Figure 5-2: LLVM implementation of `int.__add__`.

Lower level operations like addition and subtraction often must be expressed in a lower level representation. Rather than add specialized instructions for these operations, we enable functions to be written in raw LLVM IR, akin to how assembly can be embedded in C++ code. This approach enables low-level operations to be written without interfacing directly with the compiler. We show an example of this technique in use by implementing the `int.__add__` function in Figure 5-2 [32].

### 5.1.2 A Basic Example

For the remainder of this thesis, we show CIR code using a Lisp-like representation. Figure 5-3 [32] contains the CIR equivalent of the code in Figure 3-1. In particular, note that the function becomes a `BodiedFunc` with one `int` argument and no variables. As explained above, the `if` statement becomes a flow, with each side having its own `ReturnInstr` containing either a constant or magic function call.

### 5.1.3 Bidirectionality

As shown in Chapter 4, Codon makes extensive use of type and function specializations due to its LTS-DI type checker. This raises an interesting problem for CIR:

```
(bodied_func
  '"fib[int]"
  (type '"fib[int]")
  (args (var '"n" (type '"int") (global false)))
  (vars)
  (series
    (if (call '"int.__lt__[int,int]" '"n" 2)
      (series (return 1))
      (series
        (return
          (call
            '"int.__add__[int,int]"
            (call
              '"fib[int]"
              (call '"int.__sub__[int,int]" '"n" 1))
            (call
              '"fib[int]"
              (call '"int.__sub__[int,int]" '"n" 2))))))))
```

Figure 5-3: CIR equivalent of Fibonacci function.

nearly every type and function is generic, making raw manipulation of the IR difficult. As a motivating example, consider a simple transformation that replaces multiplications by a power-of-2 with left shifts. A naive approach would simply look up the `__mul__` magic method, check for a constant power-of-2 multiple, and replace invocations with a `__lshift__` call. Unfortunately, however, the method may not even exist as LTS-DI instantiates types and functions lazily.

To address this issue, we introduce the notion of a *bidirectional* IR, wherein passes can easily interact with the type checker to generate new instantiations of types and functions. Beyond enabling parity with other IRs, this approach has numerous benefits:

- *Language features and optimizations can be implemented in Codon itself.* Several optimizations in Seq [37] involve constructs impractical to implement in either IR or LLVM. For example, the prefetch optimization involves a dynamic scheduler. CIR's bidirectionality enables language developers to offload most implementation effort to Codon source.

- *Passes can generate new instantiations of types.* Consider an optimization that requires a list type: the pass can simply instantiate the `List` type as needed. This can be a complex process as it requires fully invoking the type checker and potentially realizing many new functions.

- *Passes can utilize Codon's expansive type system.* IR passes can be generic and operate over many types. Importantly, this does not require coding complex rules for every target type.

This bidirectionality is implemented by maintaining some level of correspondence between IR and AST objects. In particular, types, though fully realized and non-generic, maintain references to their corresponding AST types. At pass time, developers can use these references to query for generics or to realize new types. For example, a simple `UInt8` type can be inspected to find the width generic (in this case 8) and used to realize a new `List[UInt8]`. Similarly, users can inspect the generics of functions and realize new instantiations from different input types.

Figure 5-4 contains an implementation of the power-of-2 transformation discussed above; we make use of Codon's bidirectionality to look up and instantiate the shift-left method. Far from being a convenience feature, though, CIR's bidirectionality enables many classes of optimizations that would not otherwise be possible without substantial effort. In subsequent chapters of this thesis, we overview many optimizations implemented using this approach.

### 5.1.4 Extensibility

Due to Codon's flexibility and bidirectional IR, large portions of DSL implementation can be conducted in the source itself. For example, Seq implements its custom types and large portions of its optimizations in Codon source. This allows DSL code to be intrinsically interoperable, with both the framework and other DSLs. To make use of DSL code, we propose a plugin system to allow developers to bundle Codon source with dynamic libraries. At runtime, the Codon compiler simply loads the library, calling the appropriate hooks to register passes and analyses.

```
struct InspectionResult {
  bool valid;  // true if pattern matched
  Value *val;  // value being multiplied
  int64_t mul; // multiplier, must be power-of-2
};

class PowOfTwoOptimization : public OperatorPass {
  void handle(CallInstr *v) {
    auto *M = v->getModule();

    // inspectCall not shown:
    // 1) checks for pattern
    // 2) extracts value and constant
    InspectionResult r = inspectCall(v);
    if (!r.valid) return;

    auto *type = r.val->getType();
    auto *shl = M->getOrRealizeMethod(type,
                                      Module::LSHIFT_MAGIC_NAME,
                                      {type, M->getIntType()});
    v->replaceAll(util::call(shl, {r.val,
                                   M->getInt(log_2(r.mul))}));
};
```

Figure 5-4: C++ implementation of a multiply-by-power-of-2 transformation.

On the IR level, Codon takes a different approach for customization than other DSL frameworks like MLIR [20]. Many of these frameworks enable customization on all aspects of the IR; this enables considerable flexibility. For a tool built with compatibility and ease of use in mind, however, this adds complexity in terms of integrating with existing or new passes and analyses. As such, Codon restricts customization to a few narrow categories: flows, types, instructions, and constants (see Table 5.2). DSL writers simply subclass the appropriate node (e.g. `CustomType`) and interact with the remainder of the framework declaratively. For example, custom constants must expose an LLVM "builder" to enable the runtime to construct the appropriate bytecode.

We show an example of a custom type — a single-precision floating point type — in Figure 5-5 [32]. The type simply inherits from the `CustomType` class and implements

| | Description | Examples |
|---|---|---|
| `CustomType` | DSL type. | Multidimensional arrays |
| `CustomConst` | DSL constant. | Single-precision float constant. |
| `CustomFlow` | DSL control flow structure. | Parallel loop. |
| `CustomInstr` | DSL instruction. | Multidimensional array access. |

Table 5.2: Listing of CIR customizable nodes.

the `getBuilder` function. This allows the Codon framework to build the appropriate LLVM type at compile time.

### 5.1.5 Code Generation

Codon uses LLVM to generate native code. The conversion from Codon IR to LLVM IR is generally a straightforward process, following the mappings listed in Table 5.1. Most Codon 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. Codon 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, most loops iterates over a generator (e.g. `for i in range(10)` iterates over the `range(10)` generator). Hence, it is critical that generators in Codon carry no extra overhead, and compile to equivalent code as standard C `for`-loops whenever possible. To this

```
class Float32TypeBuilder : 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 AcceptorExtend<CustomType> {
  unique_ptr<TypeBuilder> getBuilder() const {
    return make_unique<Float32TypeBuilder>();
  }
};
```

Figure 5-5: Implementation of 32-bit float type.

end, Codon 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 Codon'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.)

Codon uses a lightweight runtime library when executing code. In particular, the Boehm garbage collector [5]—a drop-in replacement for `malloc`—is used to manage allocated memory, in addition to OpenMP for handling parallelism and `libbacktrace`[2]

[1] LLVM coroutines are also used by Clang versions 6 and later to implement the C++ Coroutine Technical Specification [28].

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

for exception handling. Codon offers two compilation modes: *debug* and *release*. Debug mode includes full debugging information, allowing Codon 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.

## 5.2 Passes & Analyses

CIR provides a rich API for optimization via its `PassManager` construct. The framework enforces a distinction between *analyses*, which do not modify the underlying IR but produce a result, and *passes*, which manipulate the IR. Both operations can be easily implemented using various builtin utilities. In Figure 5-6, we show an example of an integer constant folding pass implemented using the `OperatorPass` utility, which automatically traverses the entire structure. Since Codon represents binary operations as function calls, this pass simply recognizes appropriate function calls and replaces them with the calculated value.

Complex passes can make easy use of CIR's bidirectionality to create new types and functions. This allows passes to be general and operate over many different types, all the while taking advantage of Codon's rich type system. Additionally, passes can make use of a flexible *match* idiom, which allows pattern matching on entire IR tree structures. We show full examples of these features in action in Chapter 6. Further, we add a detailed description of important utilities in Appendix B.

## 5.3 Previous Design

Naturally, this intermediate representation has gone through many evolutions. Initially, we implemented a rigid, completely flattened representation similar to that of Rust's IR [24]. Rather than maintain a control-flow hierarchy, the entire program

```
struct InspectionResult {
  bool valid;    // true if pattern matched
  string magic; // the magic name
  int64_t lhs;  // value of the left
  int64_t rhs;  // value of the right
};

class IntFolder : public OperatorPass {
  void handle(CallInstr *v) {
    auto *M = v->getModule();

    // inspectCall not shown:
    // 1) checks for pattern
    // 2) extracts magic name and constants
    InspectionResult r = inspectCall(v);
    if (!r.valid) return;

    if (r.magic == Module::ADD_MAGIC_NAME) {
      v->replaceAll(M->getInt(r.lhs + r.rhs));
    } else if (r.magic == Module::MUL_MAGIC_NAME) {
      v->replaceAll(M->getInt(r.lhs * r.rhs));
    } else if (r.magic == Module::SUB_MAGIC_NAME) {
      v->replaceAll(M->getInt(r.lhs - r.rhs));
    } else if (r.magic == Module::FLOOR_DIV_MAGIC_NAME) {
      v->replaceAll(M->getInt(r.lhs / r.rhs));
    }
  }
};
```

Figure 5-6: Implementation of a constant folding pass.

was converted into basic blocks linked together with *terminators*. Within each block, instructions expressed the core computation, each comprised of an optional *l-value*, or memory location, and an *r-value*. Each r-value was itself composed of *operands*. We show a full listing of this structure in Figure 5-7 and Table A.7 (found in Appendix A).

This design had numerous shortcomings. Its lack of value nesting forced the IR generation step to create an excessive number of temporary variables. This is extremely inefficient from a compilation perspective, forcing LLVM to attempt to promote many temporary variables to SSA values. More concerningly, however, it

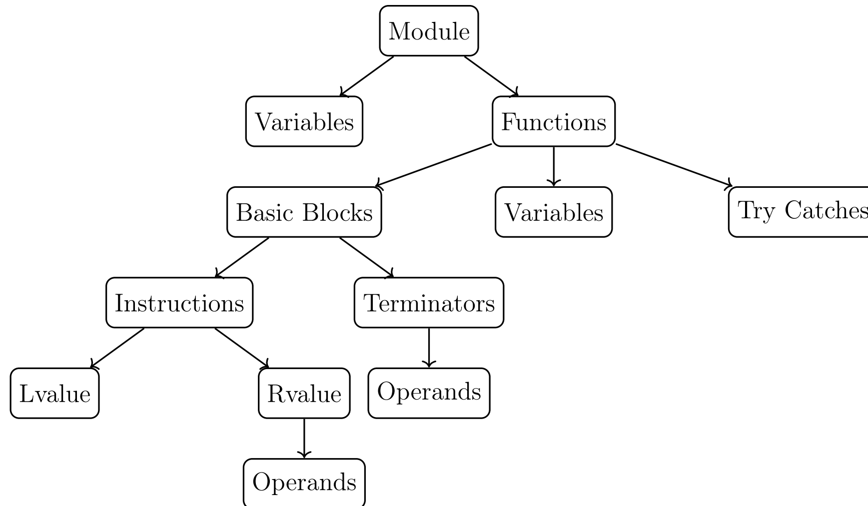


Figure 5-7: Legacy CIR structure.

made analyses and passes extremely difficult. Since the structure was completely disjoint from the original structure, recognizing patterns was challenging, requiring complex analyses even for the most simple cases.

Consider, for example, a simple loop analyses that recognizes the number of iterations of classic Pythonic `for i in range(<const>)` loops. In the previous design, such a pass would have to traverse the entire basic block graph, detect cycles, find the loop condition, and decipher multiple temporary variables to find the `range` generator. This generator must then be analyzed to extract the constant. In the new IR, the pass can simply inspect the `for` flow's generator.

# Chapter 6

# Creating DSLs with Codon

In this chapter, we explore Codon's performance on a variety of Pythonic tasks, with and without optimizations. Then, we showcase various DSLs implemented using the framework.

## 6.1 Python

Beyond implementing new DSLs, Codon can be used to vastly accelerate existing Python programs.

### 6.1.1 Standard Codon

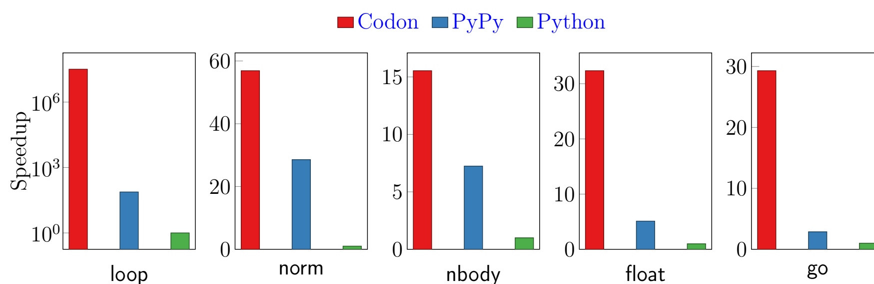


Figure 6-1: Performance comparison on various Pythonic benchmarks.

To evaluate this impact, we show Codon's runtime on a variety of benchmarks derived from the standard Python performance testing suite[1]. Since the original Seq [37]

[1] https://github.com/python/pyperformance

paper demonstrated increased performance over other Pythonic implementations, we limit this analysis to Codon, CPython, and PyPy [35]. With minimal modification (see Appendix C for full listings), Codon demonstrates orders of magnitude better performance.

We show this comparison in Figure 6-1 [32]. For the loop benchmark, which iteratively calculates $\sum_{i=1}^{10^4-1}\sum_{j=1}^{10^4-1}(i+j)$, Codon's statically compiled nature allows the central loop to be elided into a constant, resulting in an especially dramatic speedup. Similar, albeit less dramatic, speedups occur across the benchmarks.

### 6.1.2 Dictionary Access

For many applications, dictionary querying and modification comprises a substantial portion of runtime, particularly for counting tasks. In Pythonic code, this typically manifests itself as a `d[x] = d[x] <op> <value>` pattern, which we denote the *get/set* pattern. Unfortunately, however, this is extremely inefficient, paying the cost of two dictionary lookups.

```
@extend
class Dict[K, V]:
  # __dict_do_op__ not shown
  def __dict_do_op_throws__[F, Z](self,
                                  key: K,
                                  other: Z,
                                  op: F):
    x = self._kh_get(key)
    if x == self._kh_end():
      raise KeyError(str(key))
    else:
      self._vals[x] = op(self._vals[x], other)
```

Figure 6-2: Codon component of dictionary optimization pass.

A better solution would lookup the appropriate value's index in the dictionary's internal array and perform the modification in place. To maintain generality, we implement this optimization using Codon's bidirectional IR. In Figure 6-2, we show the Codon component of this optimization, which accepts a key value, constant, and

```
struct InspectionResult {
  bool valid;  // true if pattern matched
  Func *func;  // the function being applied
  Value *dict; // the dictionary
  Value *key;  // the key
  Value *val;  // the other value
  Const *dflt; // default value, may be nullptr
};

class GetSetOptimization : public OperatorPass {
  void handle(CallInstr *v) {
    auto *M = v->getModule();

    // inspectCall not shown:
    // 1) checks for pattern
    // 2) verifies that key is not a function call
    // 3) extracts and clones values
    InspectionResult r = inspectCall(v);
    if (!r.valid) return;

    Func *func;

    // Call non-throwing version if default is supplied
    if (r.dflt) {
      replacementFunc = M->getOrRealizeMethod(
            r.dict->getType(), "__dict_do_op__",
            {r.dict->getType(), r.key->getType(),
             r.val->getType(), r.dflt->getType(),
             r.func->getType()});
    } else {
      replacementFunc = M->getOrRealizeMethod(
            r.dict->getType(), "__dict_do_op_throws__",
            {r.dict->getType(), r.key->getType(),
             r.val->getType(), r.func->getType()});
    }

    vector<Value *> args = {r.dict, r.key, r.val};
    if (r.dflt)
      args.push_back(r.dflt);

    v->replaceAll(util::call(func, args));
  }
};
```

Figure 6-3: Simplified C++ component of dictionary optimization pass.

```
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 6-4: Codon implementation of word-counting benchmark.

a function. At pass time (as can be seen in Figure 6-3), we recognize the *get/set* pattern and replace it with a single call to a new instantiation of the appropriate method, supplying the key, constant, and appropriate magic function. Importantly, this pass can be used with any data type that implements the appropriate method.

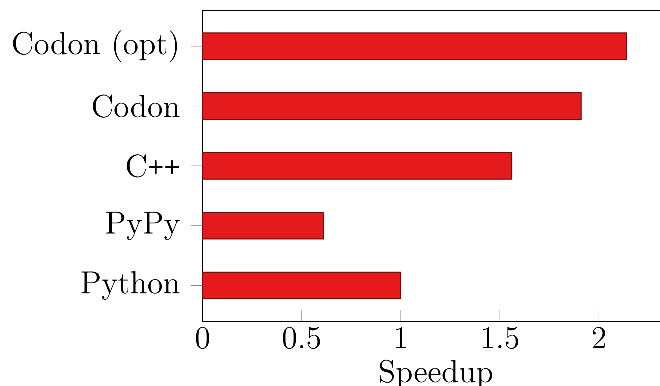


Figure 6-5: Speedup for various word-count implementations.

While this appears to be a comparatively minor optimization, this has a measurable impact for common applications. In Figure 6-4, we show the Codon implementation of a simple word-counting benchmark. When tested on 100 million lines of Wikipedia text [12], Codon, with and without the optimization, achieved a measurable performance increase over Python and C++ (see Appendix C for implementations in other languages). This speedup, as can be seen in Figure 6-5 [32], was especially dramatic between PyPy and the optimized Codon implementation, with the optimization giving a noticeable advantage.

### 6.1.3 String Manipulation & I/O

```
void StrAdditionOptimization::handle(CallInstr *v) {
  auto *M = v->getModule();

  auto *f = util::getFunc(v->getCallee());
  if (!f || f->getUnmangledName() != "__add__")
    return;

  InspectionResult r;
  inspect(v, r);

  if (r.valid && r.args.size() > 2) {
    vector<Value *> args;
    util::CloneVisitor cv(M);

    for (auto *a : r.args) {
      args.push_back(cv.clone(a));
    }

    auto *arg = util::makeTuple(args, M);
    args = {arg};
    auto *replacementFunc =
        M->getOrRealizeMethod(M->getStringType(), "cat",
                              {arg->getType()});
    seqassert(replacementFunc,
              "could not find cat function");
    v->replaceAll(util::call(replacementFunc, args));
  }
}
```

Figure 6-6: Simplified C++ implementation of the string addition folder.

In Python, the most performant method to add strings is with a single `str.cat` call. Often, however, developers implement this operation with large string addition trees. While easy-to-understand and idiomatic, this approach wastes resources by creating an excessive number of temporary strings (one per addition). As such, we implement a simple pass, shown in Figure 6-6. This pass simply checks for addition trees and collapses their arguments into a tuple, which it supplies to a new instantiation of the `str.cat` method.

We take a similar approach for optimizing IO calls. Since both addition and

format strings collapse into `str.cat` calls, we optimize file writing and printing by eliminating the temporary string created by the concatenation. In applications with extensive logging, eliding temporary strings can noticeably lower memory pressure, providing a potential performance benefit for non-IO bound tasks.

## 6.2 Seq

Seq [37], the inspiration for the Codon project, adds many features to Codon's core. The language adds "sequence" and "$k$-mer" (or length $k$ substrings) types to represent genomic data. These types are augmented with a new standard library and biology-specific optimizations. Since many genomics computations are easily implemented using Codon's pipeline syntax, we concentrate on optimizing them to reduce memory access costs and superfluous computation.

### 6.2.1 Custom Nodes

Seq provides an optimized reverse compliment function for $k$-mers [37]. Since this operation is non-trivial to implement in Codon source, we define a `KmerRevcomp` instruction; while this operation is not emitted by the parser, we define a corresponding `KmerRevcompInterceptor` pass to convert `__invert__` calls.

### 6.2.2 Prefetch Optimization

Many genomics applications involve repeated lookups into large data structures such as FM-indices. Since these accesses can be random, cache misses take up a substantial portion of runtime. As such, we add a `@prefetch` annotation to instruct the compiler to overlap cache misses in a pipeline. We show a simpl

Consider a simple pipeline `inputs |> search`, where `search` is a simple function with the prefetch annotation. Rather than simply iterate over `inputs` and run `search` for each value, the compiler converts the `search` function to a coroutine that prefetches from the appropriate structure and then yields. These coroutines are man-

```
@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: Codon implementation of the dynamic scheduler.

aged by a dynamic scheduler, shown in Figure 6-7 [32] and implemented in Codon itself, that resumes the coroutines when their inputs are ready.

A simplified implementation of this function transformation can be found in Figure 6-8 [32]. We benchmarked this optimization using a simple indexing task (shown in Appendix C). As can be seen in Figure 6-9 [32], this optimization results in a large speedup, particularly with larger $k$, or $k$-mer size.

### 6.2.3 Inter-align Optimization

We implement a similar optimization for alignment, a common task in genomics that lines up sequences based on overlapping bases. Conventional alignment tools operate on single pairs of sequence. To improve performance, we provide an `@interalign`

```cpp
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 6-8: Simplified C++ implementation of the prefetch function transformation.

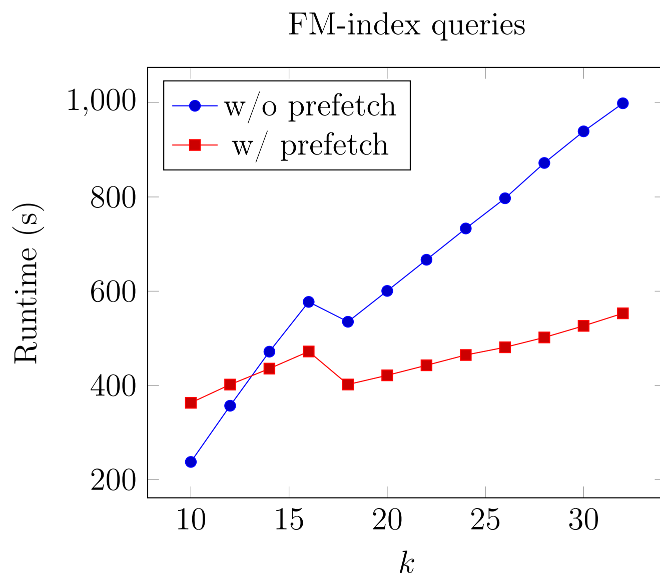


Figure 6-9: Runtimes for FM queries at various $k$.

annotation that instructs the compiler to convert the functions into coroutines as in the above optimization. Rather than prefetching however, the compiler batches sequences waiting for alignment so that they can be aligned using vector instructions.

In the spirit of the Pythonic optimizations, we also add an transformation to recognize common pipeline patterns and consolidate them into single function calls. For example, it is common to arrange a `kmers` stage before a `revcomp` stage. These can be combined into a single `_kmers_revcomp` stage, giving a noticeable performance increase with $k > 5$ [38].

## 6.3 Sequre

Secure multi-party computation (MPC) [9] is a conceptually simple class of protocols that splits participants into data owners and computing parties. Data owners share their data with computing parties such that no individual can recover the original. After some communication with peers, the computing parties then share their sub-results with the owners, who determine the final result.

The security of such protocols depends on splitting data into *shares*. For some operations such as addition and subtraction, shares can be determined without com-

munication. Unfortunately, more complex arithmetic operations such as multiplication depend on communication-heavy, high-overhead protocols like Beaver partitioning [3]. Optimizations like *Beaver partition caching* and *polynomial evaluation* [7] aim to reduce this overhead by decreasing computation; however, these approaches are challenging to implement and increase code complexity.

Sequre is a Codon DSL that aims to solve the complexity problem by combining a batteries-included MPC library with robust compiler support. In particular, Sequre adds two major CIR passes: *beaver optimization* and *polynomial optimization*. With minimal annotation through the `@secure`, these passes automatically transform code into its optimized secure MPC equivalent, vastly reducing communication overhead.

## 6.4 CoLa

Block-based approaches form the backbone of many important applications, such as image and video compression, yet lack robust support in typical programming languages. Indeed, new versions of compression algorithms, despite having many similarities, often opt to re-implement themselves from scratch, resulting in complex, large codebases. This results from a fundamental disconnect between the features provided by C/C++ (and other lower level languages) and the features necessary for concise, reusable implementations of compression algorithms. CoLa, or the **Co**mpression **La**nguage, derives from Codon to provide a better interface through purpose-built abstractions, particularly in the areas of data representation, data traversals, and data partitioning.

CoLa provides native, first-class datatypes called `Views` and `Blocks` that are multidimensional by design. Though they have no equivalent in the core Codon standard library, they are both expressive and simply implemented in Codon source.

Traversals define what order individual units of CoLa's data types are visited. Often, these patterns are difficult to define with simple loops. As such, CoLa adds special syntax, specifically the `@traversal` annotation, which instructs the compiler that a function defines traversal. Such functions contain the `tparams`, `rrot`, `rstep`,

and `link` keywords, which declaratively specify the pattern and step of the traversal. At compile-time, CoLa recognizes these keywords and desugars the declaration into a potentially complex loop nest.

# Chapter 7

# Standard Analyses & Passes

This chapter discusses various builtin Codon analyses and passes.

## 7.1 Analyses

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

### 7.1.1 Control-flow Graphs

While CIR'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 CIR. This approach, akin to that taken by Taichi [16], enables analyses to operate over a fully simplified structure without eliminating CIR's nested structure too early. Importantly, the framework can automatically construct control-flow graphs (CFGs) for all builtin nodes; as with LLVM builders, custom nodes must define CFG builders to deconstruct themselves. We package this tool into an analysis that can be registered in the `PassManager`.

As in the classic formulation, we define basic blocks, which contain values, and edges between these blocks [8]. A standard `if/else` flow, for example, will be deconstructed into 2 blocks each leading to an "end" block. This structure is particularly

useful because each value in a given block is guaranteed to execute without interruption. Therefore, analyses can often operate on blocks as a whole without having to consider individual instructions.

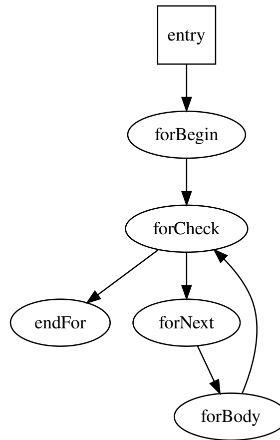


Figure 7-1: Example `for` loop control-flow graph.

In Figure 7-1, we show the control-flow of a Codon function containing a single `for` loop. Codon defines control-flow graph conversion methods for all builtin nodes. Custom nodes define builders to destruct themselves into graphs, allowing dependent analyses to work without further intervention.

### 7.1.2 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 [8]. 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.

Since custom nodes are supported by the control-flow analyses, reaching definitions works as expected. We similarly package it into a `PassManager` analysis that depends on the CFG construction. Users can query the definitions of variables at any point in the CIR structure. We expect that this analysis will be useful for constant propagation and other classical optimizations.

### 7.1.3 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 analyses. 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.

## 7.2 Passes

Codon provides standard transformations such as dead-code elimination. Though the LLVM [19] back-end provides many of the same optimizations, incorporating them as CIR passes expands the reach of domain-specific passes with minimal performance cost (less than a second for a large program). For example, a pass that depends on constants will be more useful if folding and propagation is run first.

### 7.2.1 Loop Lowering

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

Figure 7-2: Simplified CIR equivalent of a `for` loop.

In implementing the CoLa DSL (see Chapter 6), we found that Codon's canonical loop format can be cumbersome for some optimizations. Consider the simple loop `for i in range(10)`, which maps to the CIR in Figure 7-2. 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 7-3. The pass simply checks for a `__iter__` call applied to a `range`, extracts the arguments, and creates a new 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 7-3: Simplified C++ implementation of `for` loop lowering.

This is an example of "progressive lowering" [20]: the original loop form could be useful to certain passes, while the lowered form may be more helpful for others. In particular, we expect that the `ImperativeForFlow` will be helpful for loop unrolling.

### 7.2.2 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 7-4: 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 following constant propagation. Similarly, we resolve control flow (i.e. `IfFlows` and `TernaryInstr`) at compile-time for constant conditions.

```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 7-4: Simplified C++ implementation of constant propagation.

# Chapter 8

# Related Works

This chapter addresses relevant other works, specifically related to dynamic languages and Codon's type checker/IR.

## 8.1 Augmenting Dynamic Languages

Many tools have been developed to port some of the advantages of static languages, in particular performance and early error reporting, to dynamic languages. Typically, these approaches rely on external components like type checkers or re-implementations of the original interpreters and runtimes.

### 8.1.1 Type Checking

For Python, Mypy [21] is the preeminent static type checker, relying on the language's builtin type annotation syntax (identical to that of Codon). While Mypy has a degree of bidirectionality, this is limited to single statements. As a result, common cases like empty list literals are undecidable; consequently, Mypy relies on frequent use of unknown types (`Any`), which are determined at runtime. Additionally, Mypy runs completely separately from the Python interpreter, making its use optional.

Other tools like InferDL [17], Hummingbird [34], and PRuby [13] apply similar techniques to the Ruby language. In particular, these type checkers make use heuris-

tics to apply constraints to unknown types, which are then resolved. Unfortunately, these approaches are still completely outside the language itself and can result in unknown types, making the performance of statically-typed languages unattainable. Conversely, Codon integrates its type checker throughout its compiler.

### 8.1.2 Performance Optimization

Achieving high performance in Python can often re-implementing code hotspots in C/C++ and calling the native code. Short of this, the Python ecosystem also offers several frameworks and alternative interpreters to improve performance. Cython [4] compiles a restricted subset of Python, converting it to C; while this achieves an impressive speedup, it unfortunately cannot achieve full coverage of Pythonic code and still relies on the high-overhead Python runtime. PyPy [35] is an alternative interpreter that achieves some speed benefits. Like other alternative interpreters, however, its speed gains are limited by Python's dynamic nature and runtime.

## 8.2 DSL Frameworks

Many other frameworks for creating DSLs rely on embedding in strongly-typed languages, using meta-programming to create the illusion of custom syntax. Often, such DSLs are legal examples of their parent languages, compilation involving running the code and generating a new representation. One such framework is Delite [6], which embeds in Scala. Rather than run directly, Delite code — itself valid Scala — generates an intermediate representation, which DSL specific "re-write rules" and "traversals" can operate on. Similarly, AnyDSL [22] embeds in another language called Impala. For both frameworks, this IR is then converted to the target representation before being run. While this technique obviates the need for a custom frontend, developers are intrinsically limited by the syntax of the base language: adding new patterns can be difficult to impossible. Further, this creates a large gap between the optimization layer and the language source, minimizing expressiveness and restricting the optimization space. Accordingly, users of these frameworks are unable to take

advantage of their parent languages' type systems.

A similar approach can also be taken in dynamic languages, including for Python. MetaDSL [18] aims to bridge disparate Python libraries into a cohesive API. In particular, it uses Python source to generate an internal representation, which is then converted to the desired execution format. Still other languages like Racket [11] include first-class support for this paradigm, with users even allowed to mix DSLs at will. This approach can be advantageous as it only requires users to create thing translation layers. Unfortunately, however, optimization can be difficult as the DSLs are essentially sugar. Further, these languages often suffer from lower performance.

These dynamic DSLs can be improved by using higher-performance backends, which can often JIT the source to achieve better performance. For Racket, Sham [39] adds higher performance library functions that generate LLVM IR. Thus, DSL creators in Sham simply write Racket DSLs, but rewrite their code generation steps to employ Sham functions. While this achieves better performance, this approach does not allow for easy optimization, relying on an AST for compilation.

## 8.3 Intermediate Representations

Codon's IR takes inspiration from many other successful IRs. In particular, CIR's structurally resembles the IRs of Taichi [16] and Suif [40] which have hierarchical structures. In contrast, however, we use a far more restricted set of nodes. In that sense, CIR resembles LLVM IR [19] and Rust's [24] IR. CIR differs from other extensible IRs like MLIR [20] in terms of its strategy: while other frameworks enable customization at nearly all levels, CIR restricts users to extending a limit subset of features. This allows Codon to be intrinsically composeable.

# Chapter 9

# Conclusion

In this thesis, we have introduced Codon, a framework for implementing peformant Pythonic DSLs. The framework's novel frontend, IR, and builtin passes allow for easy extension and customization with minimal developer effort. These extensions come with minimal performance cost, Codon often running faster than even C++. Most importantly, however, Codon provides a model for applying high-performance methodologies, typically reserved for static languages, to dynamic source.

## 9.1 Future Work

Future work will concentrate on increasing the amount of Python code immediately compatible with Codon. In particular, we plan to add union types and inheritance. On the IR side, we hope to develop additional builtin transformations and analyses, all the while expanding the reach of existing passes. As far as library support, we plan to port existing high-performance Python libraries like NumPy [14] to Codon; this will allow Codon to become a drop-in replacement for Python in many domains. Finally, we hope to deprecate the existing OCaml parsing layer, replacing it with a hand-written alternative.

# Appendix A

# Supplemental Listings

## A.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 A.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 A.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 A.3: Listing of AST expressions.

## A.2 CIR 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 A.4: Listing of builtin CIR types.

| | **Type** | **Description** |
|---|---|---|
| `Var` | Variable | Global or local variable. |
| `BodiedFunc` | Function | CIR function. |
| `LLVMFunc` | Function | Function implemented in LLVM IR. |
| `ExternalFunc` | Function | Function implemented in library. |
| `InternalFunc` | Function | Function implemented in compiler. |

Table A.5: Listing of CIR 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 A.6: Listing of CIR values.

## A.3 Legacy CIR Nodes

| | Type | Description |
|---|---|---|
| `AssignInstr` | Instruction | Stores the r-value into the l-value. |
| `RvalueInstr` | Instruction | Runs the r-value. |
| `VarLvalue` | L-value | Variable's memory location. |
| `VarMemberLvalue` | L-value | Member's memory location. |
| `MemberRvalue` | R-value | Value of member. |
| `CallRvalue` | R-value | Result of function call. |
| `OperandRvalue` | R-value | Value of operand. |
| `StackAllocRvalue` | R-value | Value of a stack-allocated array. |
| `PartialCallRvalue` | R-value | Value of a partial call. |
| `MatchRvalue` | R-value | Result of a pattern match. |
| `PipelineRvalue` | R-value | Result of a pipeline. |
| `VarOperand` | Operand | Value of a variable. |
| `VarPointerOperand` | Operand | Location of a variable. |
| `LiteralOperand` | Operand | Constant. |
| `JumpTerminator` | Terminator | Unconditional branch. |
| `CondJumpTerminator` | Terminator | Conditional branch. |
| `ReturnTerminator` | Terminator | Return. |
| `YieldTerminator` | Terminator | Yield. |
| `ThrowTerminator` | Terminator | Throws exception. |
| `AssertTerminator` | Terminator | Asserts a condition. |

Table A.7: Listing of legacy CIR instructions, l-values, r-values, operands, and terminators.

# Appendix B

# CIR Utilities

Codon provides several useful utilities for manipulating IR. In this section, we show several relevant utilities that are used in the body of this thesis.

## B.1 IR Manipulation

| Method | Description |
|---|---|
| N<T> | Constructs and registers an IR node. |
| getOrRealizeMethod | Gets or realizes a method with a parent class. |
| getOrRealizeFunc | Gets or realizes a function. |
| getOrRealizeType | Gets or realizes a type. |
| get{primitive}Type | Gets or the appropriate type. |
| get{primitive} | Gets a constant of the appropriate type. |

Table B.1: Listing of select Module utilities.

| Method | Description |
|---|---|
| replaceAll | Replaces all instances of the value. |
| getUsedValues | Gets all children values. |
| getUsedTypes | Gets all children types. |
| getUsedVars | Gets all children variables. |

Table B.2: Listing of select Node utilities.

<table>
<tr><th>Method</th><th>Description</th></tr>
<tr><td>operator==</td><td>Calls equality magic.</td></tr>
<tr><td>operator!=</td><td>Calls inequality magic.</td></tr>
<tr><td>operator<</td><td>Calls comparison magic.</td></tr>
<tr><td>operator></td><td>Calls comparison magic.</td></tr>
<tr><td>operator<=</td><td>Calls comparison magic.</td></tr>
<tr><td>operator>=</td><td>Calls comparison magic.</td></tr>
<tr><td>operator+</td><td>Calls addition magic.</td></tr>
<tr><td>operator-</td><td>Calls subtraction magic.</td></tr>
<tr><td>operator/</td><td>Calls division magic.</td></tr>
<tr><td>operator%</td><td>Calls modulo magic.</td></tr>
<tr><td>pow</td><td>Calls power magic.</td></tr>
<tr><td>operator»</td><td>Calls shift magic.</td></tr>
<tr><td>operator«</td><td>Calls shift magic.</td></tr>
<tr><td>operator|</td><td>Calls or magic.</td></tr>
<tr><td>operator&</td><td>Calls and magic.</td></tr>
<tr><td>operator^</td><td>Calls xor magic.</td></tr>
<tr><td>operator||</td><td>Generates boolean or.</td></tr>
<tr><td>operator&&</td><td>Generates boolean and.</td></tr>
<tr><td>operator[]</td><td>Calls get item magic.</td></tr>
<tr><td>toInt</td><td>Calls int magic.</td></tr>
<tr><td>toBool</td><td>Calls bool magic.</td></tr>
<tr><td>toStr</td><td>Calls string magic.</td></tr>
<tr><td>len</td><td>Calls length magic.</td></tr>
<tr><td>iter</td><td>Calls iter magic.</td></tr>
</table>

Table B.3: Listing of select `Value` utilities.

## B.2 Miscellaneous Tools

| Visitor | Description |
|---|---|
| `Operator` | Visits all values in a program. |
| `CloneVisitor` | Clones CIR nodes. |
| `FormatVisitor` | Formats CIR nodes to a string. |
| `match` | Compares two CIR nodes. |

Table B.4: Listing of miscellaneous utilities.

# Appendix C

# Benchmark Information

All benchmarks 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.

## C.1 Pythonic Benchmarks

### C.1.1 PyPerformance

As in [32], we use benchmarks adapted from Python's benchmark suite (`github.com/python/pyperformance`):

- 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.fpy`): $n$-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.

- 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.

The loop benchmark was taken from realpython.com/pypy-faster-python, and is a simple double for-loop. Unlike the other benchmarks, this one was timed within the code itself due to its speed in Codon.

### C.1.2 Word Count

```
int main(int argc, char *argv[]) {
    cin.tie(nullptr);
    cout.sync_with_stdio(false);

    if (argc != 2) {
        cerr << "Expected one argument." << endl;
        return -1;
    }

    ifstream file(argv[1]);
    if (!file.is_open()) {
        cerr << "Could not open file: " << argv[1] << endl;
        return -1;
    }

    unordered_map<string, int> map;
    for (string line; getline(file, line);) {
        istringstream sin(line);
        for (string word; sin >> word; )
            map[word] += 1;
    }

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

Figure C-1: C++ implementation of word-counting benchmark.

## C.2 Seq Benchmarks

### C.2.1 Prefetch

Originally shown in [32]:

```
import sys, bio.fmindex
from bio import *

total = 0
def add(count):
    global total
    total += count

@prefetch
def s(s, index):
    intv = index.interval(s[-1])     # initialize interval
    s = s[:-1]                       # trim last base
    while s and intv:
        intv = index[intv, s[-1]]    # extend interval
        s = s[:-1]                   # trim last base
    return len(intv)                 # number of hits

idx = bio.fmindex.FMIndex(argv[1])  # create the index
FASTQ(argv[2]) |> seqs |> split(k=20, step=20) |> s(idx) |> add

print(f'found {total} matches')
```

Figure C-2: Seq prefetching benchmark.

# Bibliography


[1] 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.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools.* Pearson/Addison Wesley, 2007.

[3] 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.

[4] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

[5] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, September 1988.

[6] 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.

[7] Hyunghoon Cho, David J. Wu, and Bonnie Berger. Secure genome-wide association analysis using multiparty computation. *Nature Biotechnology*, 36(6):547–551, Jul 2018.

[8] Keith Cooper and Linda Torczon. *Engineering a compiler.* Elsevier, 2012.

[9] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing.* Cambridge University Press, 2015.

[10] Luis Damas. Type assignment in programming languages. *KB thesis scanning project 2015*, 1984.

[11] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. https://racket-lang.org/tr1/.

[12] Wikimedia Foundation. Wikimedia downloads, 2021.

[13] 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.

[14] 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.

[15] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[16] 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.

[17] 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.

[18] Quansight Labs. Metadsl, 2021.

[19] 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.

[20] 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.

[21] Jukka Lehtosalo. Mypy.

[22] 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.

[23] Manas. Crystal.

[24] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[25] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[26] Robin Milner and Mads Tofte. The definition of standard ml. 1990.

[27] Nevena Milojković, Mohammad Ghafari, and Oscar Nierstrasz. It's duck (typing) season! In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, page 312–315. IEEE Press, 2017.

[28] Gor Nishanov. ISO/IEC TS 22277:2017, Dec 2017.

[29] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[30] François Pottier and Yann Régis-Gianas. Menhir.

[31] 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.

[32] Gabriel Ramirez, Ariya Shajii, Jessica Ray, Haris Smaljović, Bonnie Berger, Ibrahim Numanagić, and Saman Amarasinghe. Codon: Creating High-Performance, Pythonic DSLs. Submitted to OOPSLA 2021.

[33] Fabrice Rastello. *SSA-based compiler design*. Springer, 2010.

[34] 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.

[35] Armin Rigo and Samuele Pedroni. Pypy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 944–953, New York, NY, USA, 2006. Association for Computing Machinery.

[36] 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.

[37] 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.

[38] Ariya Shajii, Ibrahim Numanagić, Alexander T. Leighton, Haley Greenyer, Saman Amarasinghe, and Bonnie Berger. A python-based optimization framework for high-performance genomics. *bioRxiv*, 2020.

[39] Rajan Walia, Chung chieh Shan, and Sam Tobin-Hochstadt. Sham: A DSL for Fast DSLs, 2020.

[40] 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.