LEAVE THE FEATURES: TAKE THE CANNOLI

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Jonathan Catanio

June 2018

COMMITTEE MEMBERSHIP

TITLE: Leave the Features: Take the Cannoli

AUTHOR: Jonathan Catanio

DATE SUBMITTED: June 2018

COMMITTEE CHAIR: Aaron Keen, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Phillip Nico, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: John Seng, Ph.D.
Professor of Computer Science

ABSTRACT

Leave the Features: Take the Cannoli

Jonathan Catanio

Programming languages like Python, JavaScript, and Ruby are becoming increasingly popular due to their dynamic capabilities. These languages are often much easier to learn than other, statically type checked, languages such as C++ or Rust. Unfortunately, these dynamic languages come at the cost of losing compile-time optimizations. Python is arguably the most popular language for data scientists and researchers in the artificial intelligence and machine learning communities. As this research becomes increasingly popular, and the problems these researchers face become increasingly computationally expensive, questions are being raised about the performance of languages like Python. Language features found in Python, more specifically dynamic typing and run-time modification of object attributes, preclude common static analysis optimizations that often yield improved performance.

This thesis attempts to quantify the cost of dynamic features in Python. Namely, the run-time modification of objects and scope as well as the dynamic type system. We introduce Cannoli, a Python 3.6.5 compiler that enforces restrictions on the language to enable opportunities for optimization. The Python code is compiled into an intermediate representation, Rust, which is further compiled and optimized by the Rust pipeline. We show that the analyzed features cause a significant reduction in performance and we quantify the cost of these features for language designers to consider.

ACKNOWLEDGMENTS

Thanks to:

- My family and friends for their incredible support
- My advisor, Aaron Keen, for his endless wisdom
- The Rust and Python communities for all of their exceptional help on the intricacies of their respective languages

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Listings

Chapter 1

INTRODUCTION

Python is a widely used language that has gained a significant amount of popularity in recent years with its support for the scientific, machine learning, and artificial intelligence communities [21, 28]. These communities rely on the dynamic type system, automatic memory management, and other "dynamic" features that make the language straightforward to learn and use. These features, although convenient, come at a cost. Python has historically been a slow language whose dynamic features tend to result in suboptimal performance [20, 22]. These abstractions, coupled with the computing needs of the aforementioned communities, can quickly become problematic as performance needs increase.

Attempts to mitigate the performance issues of the original implementation of Python, referred to as CPython, have been made by various open source communities. One such project is PyPy [23], which provides an implementation of Python written in a subset of Python called RPython, or restricted Python [2]. Their approach uses a combination of RPython and just-in-time compilation to produce a generally better performing Python interpreter. Projects like PyPy are focused on increasing the performance of the language and not necessarily addressing the underlying language features that cause these performance issues.

The purpose of this work is to identify language features that result in performance issues and quantify the cost of including these features in a programming language. The first feature that we hypothesize to be the most detrimental to performance is run-time modification of objects and scope. The second is a combination of Python's type system and the aforementioned features that ultimately inhibit compile time type inference.

Consider the Python keyword `del` which allows the user to delete an attribute from an object or to remove an identifier from the current scope. This is illustrated in the following code snippet:

```
1  class Example:
2      def __init__(self):
3          self.x = 1.0
4          self.y = 2.0
5
6  obj = Example()
7  obj.value = 'new attribute' # adds a new attribute
8  obj.x = 'type change' # the x attribute now has a different type
9
10 del obj.x # obj no longer contains attribute x
11 del obj # obj is no longer defined in the current scope
```

Listing 1.1: Python 'del' keyword example

The definition of the `Example` class initilizes an `x` and `y` attribute on instantiation. This information could be supplied to the compiler, but these attributes could be removed (Line 10) and more attributes could be added (Line 7). Therefore, determining all attributes of an object at compile time is very difficult, perhaps impossible. Consequently, this hinders the ability to perform static analysis on Python code.

Despite the difficulties of optimizing Python code, attempts have been made. In addition to the aforementioned PyPy, another project attempted to infer atomic types of local identifiers, by adding type specific bytecodes to the compiler, with little success [4]. Alternatively, our analysis considers the removal or restriction of specific features. Therefore, we wanted to measure the performance of a language with and without the features of interest.

We developed Cannoli, a compiler for a subset of Python 3.6.5 written in the Rust

programming language, that compiles Python code into Rust code. The first iteration of Cannoli includes the dynamic features that we are interested in analyzing. The exception is the `del` keyword, which is not supported in Cannoli. Adding support for `del` would not affect the compiled code of programs that do not use the keyword, but would preclude or complicate optimizations if present. Alternative Cannoli versions apply restrictions to the language that enable the implementation of various optimization techniques at compile-time. The performance of Cannoli is compared with and without the dynamic features. A performance comparison is also done with CPython and PyPy, for context, but the focus was not to outperform these implementations, since they have been in development for many years. Instead, we suggest that the performance improvements exemplified by Cannoli might also be observed in CPython. We hope to provide further information for language designers who are considering adding or removing features discussed in this paper by quantifying the cost of those features.

The work of this thesis is divided as follows. Chapter 2 discusses the background information pertaining to the implementation of the Python compiler. Other implementations of Python and related work are discussed in Chapter 3. Chapter 4 describes the actual implementation of the compiler and its associated elements. Chapter 5 discusses the features of the language that were evaluated and optimized. Our results are presented in Chapter 6. Chapter 7 suggests future work that may be done to further develop Cannoli. Finally, Chapter 8 concludes the thesis.

Chapter 2

BACKGROUND

2.1 Dynamic vs. Static Typing

Dynamic and static typing are two different approaches for enforcing a programming language's type system. This type system refers to a set of rules that govern the assignment of types to various elements of a program. Common types are strings, integers, and floating point numbers. These types can be assigned to an identifier or variable which may be used throughout a program.

A language is considered statically typed if the types of program elements are known before run time. Examples of statically typed languages are C/C++ and Java. Statically typed languages provide the ability to enforce semantic properties described by the type system before program execution. This often results in many software bugs being caught and fixed before a program is even run. Additionally, static typing enables numerous optimization techniques that usually result in better performing code.

```
1 int add(int a, int b) {
2     return a + b;
3 }
```

Listing 2.1: Example of a statically typed function

A statically typed function is illustrated in Listing 2.1. In this case the compiler is able to validate the arguments to ensure that their types conform to the parameter types defined by the function signature. Similarly, the compiler can also validate that the return value is being used in the appropriate context. This type checking is often done as a first pass before compilation begins, thus enabling improved code

generation.

A language is considered dynamically typed if types are associated with values at run time. Named program elements are not necessarily assigned a fixed type and instead hold a value whose type will be determined at run time. Examples of dynamically typed languages are Python, Ruby, and JavaScript. Among other things, dynamic typing provides more flexibility and speed during development. Unfortunately, this makes optimization from static analysis very difficult and sometimes impossible.

```
1 def add(a, b):
2     a + b
```

Listing 2.2: Example of a dynamically typed function

Listing 2.2 illustrates a dynamically typed alternative to Listing 2.1. The dynamically typed `add` function inherently allows generic use. A caller could pass in any two types and, if the addition operator is implemented for those two types, it will successfully return. A more complex solution would need to be developed in order to replicate this functionality in the statically typed case. If two incompatible types were passed into the Listing 2.2 function the program would crash at run time.

There are other clever type systems that try to blend the two. For instance, the Rust programming language employs type inference which tries to alleviate some of the verbosity accompanied with static typing sans type inference.

```
1 fn add(a: usize, b: usize) -> usize {
2     a + b
3 }
4 fn main() {
5     let first_val = 1;
6     let second_val = 5;
7     add(first_val, second_val);
8 }
```

Listing 2.3: Example of type inference in Rust

The variables `first_val` and `second_val` in Listing 2.3 do not require a type to be explicitly specified. The Rust compiler will identify that the variable was assigned a specific type and at that point the variable will be required to maintain that type for the duration of its lifetime. Alternatively, the `add` function parameters need to have their types explicitly stated since the compiler would not be able to determine the expected types at compile-time. The benefit of static typing with type inference is that some of the verbosity associated with explicitly stating types is removed without sacrificing compile-time checks and optimizations.

Recent versions of Python have included type annotations that may be applied to variable assignments and function parameters. The original Python enhancement proposal for function annotations states that Python would not attach any significance to annotations [30]. As debates continue on the benefits of static versus dynamic typing and vice versa, one may assume that later versions of Python will leverage type annotations when possible. The Python community, as well as others, seem to be leaning toward a compromise where aspects of static and dynamic typing can be used together [19].

2.2 Python

Python is a dynamically typed language. Its behavior differs from other dynamically and statically typed languages in the way scope and mutability are defined. This section covers some of the dynamic features of the language that will be analyzed with Cannoli.

2.2.1 Scope

Scope in Python varies in level from function scoping to module scoping and finally to global scoping. Scope can be resolved by following the LEGB rules (Local, Enclosing, Global, Built-in). Variable assignments within a function are placed at the local level. That being said, a variable may only be used after it has been introduced into scope. Languages like JavaScript differ in this aspect due to a concept called "hoisting". Python supports function scope whereas languages like Java and C support block scope.

```
1  x = 1 # variable belonging to the global scope
2  def example():
3      x = 23 # local to the function overriding globally defined x
4      if True:
5          if True:
6              if True:
7                  y = 99 # still local to the function
8      print(x, y)
9  print(x)  # prints '1'
10 example() # prints '23 99'
11 print(x)  # prints '1'
```

Listing 2.4: Python's varying levels of scope

The toy example in Listing 2.4 illustrates interesting instances of scope resolution.

The variable `x` is defined in both the global scope and the function `example`'s local scope. If `x` was not defined in `example`, but still used in the `print` statement at Line 8, the interpreter would check the local scope which does not contain the mapping. It would then check any enclosing scope, though none exist in this case since `example` is not nested within another function. Finally, it would find the mapping for the identifier `x` in the global scope. In this instance there is no need to check the built-in scope because the mapping was already located in the global scope. The built-in level is reserved for the preassigned identifiers in the built-in module defined by the language, examples are: `print`, `help`, and `SyntaxError`. The variable `y` is also local to the `example` function. A run time error would occur if the expression `print(y)` was placed between Lines 3 and 4 because the local scope would not yet contain the mapping for the identifier `y`.

There are ways to force the Python interpreter to look for identifiers outside of the local-most scope. Using the keywords `global` and `nonlocal` in Python3 will signal to the interpreter to search the global scope or the next enclosing scope respectively. These keywords become useful in situations like those presented in Listing 2.5.

```
1 x = 1
2 def bad_func():
3     x += 1
4 def good_func():
5     global x
6     x += 1
7 bad_func()  # throws exception: 'UnboundLocalError'
8 good_func() # successfully increments the global variable x
```

Listing 2.5: Example of global keyword use

When Python evaluates a function, variable assignments become local to that scope. Here the augmented assignment, `+=`, in `bad_func` assigns the variable name `x`

to the current scope. Unfortunately, it is never assigned before it is referenced thus triggering a run time error. The `global` keyword is used in the function `good_func` to inform the Python interpreter to look for the identifer in the global scope.

2.2.2 Dynamic Objects

Dynamic objects can be described as the mutation of the structure of classes and objects at run time [10]. Classes and objects may have attributes added or removed during run time. An example of this is shown in Listing 2.6.

```
1  class Example:
2      x = 1
3      def __init__(self):
4          self.y = 2
5      def method(self):
6          pass
7      def static_method():
8          pass
9
10 obj = Example()      # instantiate an Example
11 obj.val = True       # adding a new attribute on-the-fly
12 del obj.val          # deleting the attribute val from obj
13 del Example.method # Example no longer has the method attribute
14 obj.method()         # Run time error: 'AttributeError'
```

Listing 2.6: Example of dynamic objects in Python

Lines 10 through 14 of Listing 2.6 provide examples of object mutation at run time. Line 11 adds a new attribute to the `obj` object but this does not add the attribute to the `Example` class. However, the removal of the `method` attribute from the `Example` class on Line 13 does affect the instantiated object. The interpreter will throw a run time exception when it encounters Line 14 because the attribute was

deleted from the base class.

The dynamic nature of Python classes and objects result in difficulties when performing static analysis. Even if the type of `obj` was known to be `Example`, the compiler cannot make any guarantees that the properties of both `obj` and `Example` will remain the same throughout the duration of the program.

2.3 Rust

The Rust programming language is a relatively new language targeting systems programming. It advertises speed, memory safety, and thread safety [9]. It also features interesting programming paradigms that differ substantially from languages like C, Java, and Python.

2.3.1 Ownership

The Rust compiler tracks ownership of values to enable automatic memory management. When ownership expires, the value is deallocated. The Rust Programming Language book, or "The Book", describes ownership as "Rust's central feature" [16]. Ownership is how Rust manages memory without needing a garbage collector, like Java and Python, and without requiring programmers to request blocks of memory and free it after it is no longer needed, as is done in C. From the book the ownership rules are as follows:

1. Each value in Rust has a variable that is called its *owner*.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

```
1 {
2     let x = String::from('example');
3     // x is moved into y and is no longer valid, y now owns the value
4     let y = x;
5     let z = 1;
6
7     // some_func takes ownership of y and y is no longer valid
8     some_func(y);
9 } // the block scope has ended and z is dropped
```

Listing 2.7: Rust ownership

Listing 2.7 details some of the common ownership behavior seen in Rust programs. Line 4 emphasizes the second ownership rule that there may only be a single owner at a time. The value bound to `x` is moved into `y` and from that point onward `x` is invalid until it is redefined. This specific example illustrates a *double free* error that is sometimes encountered in languages that force the programmer to explicitly manage memory. Since Rust handles freeing memory when the value goes out of scope, as specified in rule three, an error would occur if `x` and `y` pointed to the same memory and both went out of scope. On Line 8 an argument is passed into a function and, thus, is moved into the function's scope.

There are a few ways to request a copy of a value in Rust. For the sake of simplicity we will focus on the `Clone` and `Copy` traits. Traits are simply a way to implement behavior across many types. Similarly to interfaces in Java, traits in Rust enable commonality among types.

```
1  {
2      let s1 = String::from('example');
3      let s2 = s1.clone(); // Strings are Clone but they are not Copy
4
5      // 5 is of integer type which implements the Copy trait,
6      // typically values that are stored on the stack are Copy
7      let x = 5;
8      let y = x; // Since x is Copy the value will be copied into y
9
10     println!('{} {}', x, y); // x and y are both valid
11
12     some_func(y); // y is copied into the function parameter
13 } // the block scope has ended and x, y, s1, and s2 are dropped
```

Listing 2.8: Rust Copy trait usage

The code in Listing 2.8 shows that variables (i.e., `x` and `y`) with type implementing the `Copy` trait can be used after they are assigned to other variables or passed into functions. Lines 2 and 3 illustrate the `Clone` trait. The `String` data type is `Clone` but is not `Copy`. Since `String` data is stored in the heap rather than the stack we can obtain a deep copy of the heap data by explicitly calling the `clone` method. Therefore in our example, both `s1` and `s2` are valid after Line 3. Ownership in Rust is detailed in the Rust Programming Language book [16].

2.3.2 References and Borrowing

Ensuring that all types are `Copy` is not feasible and cloning values everywhere is not ideal for performance. In order to address these issues, references and the concept of *borrowing* are supported. The following example comes from the Rust book [16].

```
1 fn main() {
2     let s = String::from('hello');
3     let len = calculate_length(&s);
4 }
5
6 fn calculate_length(s: &String) -> usize {
7     s.len()
8 }
```

Listing 2.9: References and borrowing in Rust

In Rust the symbol `&` denotes a reference to a value while the symbol `*` denotes the opposite, *dereferencing* a value. In the listing above we see that a reference is passed to the `calculate_length` function on Line 3. This no longer moves the value of `s` into the function but instead moves a reference, thus permitting the use of `s` after the function call. Likewise, the function parameter `s` in the signature of `calculate_length` expects a reference to a `String` type and returns a `usize` representing the length of the provided string. References are considered as *borrowing* a value.

Rust employs a *borrow checker* that enforces the ownership rules from Section 2.3.1 at compilation time to ensure memory safety.

2.3.3 Mutability

Rust defaults variables to be immutable. This forces the programmer to opt into functionality that may be dangerous. Consider the example in Listing 2.10.

```
1 // Results in a compilation error
2 let x = 1;
3 x = 2;
4
5 // Successfully compiles
6 let mut x = 1;
7 x = 2;
```

Listing 2.10: Rust's opt-in mutability

The programmer must explicitly include the `mut` keyword to notify the compiler that the variable in question can be mutated. Similar principles apply to references. Mutable references are possible as long as the `mut` keyword is specified and the reference follows certain rules governed by the borrow checker. The rules for references, directly from the book, are as follows:

1. At any given time, you can have *either* but not both of:
    - One mutable reference.
    - Any number of immutable references.
2. References must always be valid (the reference cannot outlive the data).

2.3.4 References with Interior Mutability

A programmer might need to have multiple references to the same data with the ability to mutate the data. This is frequently used in the compiled Cannoli code. Rust allows this via shared references (that count the number of current references to a value) coupled with a mutable container called a `cell`. Borrow checking is deferred to run time when the `cell` containers are used. The reference borrowing rules still apply but they will only error when they are encountered during run time.

```
1  use std::rc::Rc;
2  use std::cell::RefCell;
3  fn main() {
4      let x = Rc::new(RefCell::new(0));
5      // here clone simply increases the reference count
6      let p1 = x.clone();
7      let p2 = x.clone();
8
9      *p1.borrow_mut() = 1;
10     println!('{}', x.borrow()); // prints '1'
11     *p2.borrow_mut() = 2;
12     println!('{}', x.borrow()); // prints '2'
13
14     let b1 = p1.borrow();
15     // crashes here since we already have an immutable borrow
16     let b2 = p2.borrow_mut();
17 }
```

Listing 2.11: Rust's interior mutability

The code in Listing 2.11 compiles but crashes at run time. This is because the reference rules mentioned in Section 2.3.3 are violated on Line 16 when there is an attempt to mutably borrow a reference that is currently immutably borrowed from Line 14. Before crashing, this example shows the ability to have multiple references that may request a mutable borrow on a value. This becomes extremely useful when trying to emulate functionality seen in Python like multiple variables modifying the same object or class.

Chapter 3

RELATED WORK

3.1 Jython

Jython is an implementation of the Python language for the Java platform [12]. The language currently supports Python 2.7 and work is in progress to support Python 3. Since Jython compiles to Java bytecode it is intended to run on the JVM. This provides the ability to import and use any Java class. That being said, the performance of the language is highly dependent on the speed of the JVM. Although Jython provides an alternative implementation of Python, it is not as much of an attempt to improve the performance of Python as it is to integrate the unique aspects of the Python language into the rich Java environment [13]. The work in this thesis also provides an alternate implementation of the Python language, but with the explicit goal of restricting features of the standard Python implementation to determine their performance cost.

3.2 IronPython

IronPython is another implementation of the Python language written in C# and compiles to bytecode that runs on Microsoft's .NET platform [11]. Like Jython, the biggest gain is the integration of Python into a different runtime environment, in this case, allowing programs to interact with .NET objects, libraries, and frameworks. Again, like Jython, the performance of IronPython is comparable to CPython [14].

3.3 PyPy

Perhaps most interesting is PyPy, an alternative implementation of Python (up to and including Python 3.5.3) written in a subset of Python called Restricted Python (RPython) [23, 2, 8]. The PyPy project replaced a project called Psyco which implemented a specialized just-in-time compiler for Python [24]. Unlike the aforementioned Python implementations, PyPy is a CPython compliant implementation that significantly improves the performance of the language. PyPy's goal is to create a platform that allows for easy experimentation with various virtual machine designs. This is done by separating out the semantics of a language from low-level details of its implementation, thus reducing the complexity of creating new dynamic languages [2, 18].

PyPy obtains much of its performance improvements through its tracing just-in-time (JIT) compiler. The traces are linear lists of operations that are optimized and transformed into machine code and inserted inline when necessary [3]. The tracing JIT (written in RPython) does not operate on a user program in Python but instead on PyPy's interpreter itself, allowing it to run much faster than its CPython counterpart.

PyPy is a great example of a straight performance improvement of Python but the work in this thesis diverges from their project in terms of the performance focus. PyPy acknowledges the dynamic language constructs and leverages the JIT compiler, whereas Cannoli places restrictions on dynamic features to analyze their performance cost [25]. Figures 6.1 and 6.2 show the impressive performance of PyPy compared to both Cannoli and CPython (on programs that do not use these dynamic features). These results show that their work has had a significant impact on the performance of the language without sacrificing features.

3.4 Cython

Cython is an optimizing static compiler for Python and the Cython language [5]. Cython provides the ability to write C extensions for Python in a semantically similar language. Cython translates Python into C which interacts with the CPython interpreter via Python's C API. This C code is generated as a CPython extension module that can be imported by standard Python code. Cython's most advertised feature is its support for optional static type declarations. Static type declarations provide useful hints to the compiler which generally outputs higher performing C code.

3.5 Shed Skin and Pythran

Similarly to Cannoli, Shed Skin is an experimental compiler for a restricted version of Python [6, 26]. However, it translates restricted Python 2.4-2.6 into C++ rather than a subset of Python 3.6.5 to Rust. Shed Skin requires Python programs to be implicitly statically typed, much like Cannoli's scope optimization (Section 5.1). Again like Cannoli, Shed Skin does not currently support all Python features. Shed Skin demonstrates a typical speedup of 2-200 times over CPython across 75 non-trivial benchmarks.

Pythran is a similar implementation to Shed Skin, supporting Python 2.7 that also compiles Python to C++ [7]. Pythran focuses on efficiently compiling scientific programs since it takes advantage of multi-cores and SIMD instructions. Much like Shed Skin (and objects in Cannoli), Pythran leverages static typing information provided via annotations.

Chapter 4

COMPILER

Cannoli compiles a subset of Python 3.6.5 code to Rust code. We start with an implementation that includes dynamic features. Later sections discuss changes to the compiler that restrict some dynamic features to enable optimizations. This section covers the motivation behind design decisions, implementation details, and translating Python constructs to Rust.

4.1 Why Compile Python to Rust?

We considered three different intermediate representations to compile Python code into. The first was LLVM [17], to leverage the many optimizations implemented by the LLVM compiler. Targeting LLVM, however, would require implementing a garbage collector (or simply ignoring memory management for this prototype). The implementation would be simplified by implementing a standard library to handle various elements of the language, but writing this library in LLVM was considered to not be ideal. Writing the library in C, compiled to LLVM code, would eliminate some of the complexity of writing the library directly in LLVM. Following this idea further we considered targeting C and compiling against a library written in C. Unfortunately, this does not address the issue of memory management.

Targeting Rust and compiling the output Rust code against a library written in Rust was considered the best of both worlds. We get a large number of optimizations from the Rust compiler (and the LLVM compiler) as well as memory management provided by Rust's ownership rules. Therefore, Cannoli is a Python compiler written in Rust that compiles Python code into Rust code.

4.2 Python Support

The Cannoli compiler supports a subset of Python 3.6.5. However, both the lexer and parser, which were also written in Rust, fully support Python 3.6.5. The lexer outputs an iterator of tokens and the parser outputs an abstract syntax tree as defined by the Python standard library `ast` module. The lexer and parser are both available as separate Rust modules within the Cannoli crate and may be used separately from the compiler.

Some features of the language were omitted because of time but the omitted features do not directly relate to the analysis that is done to validate this work. The main features that are not supported yet are inheritance and exceptions.

4.3 Cannolib

Cannoli includes a standard library written in Rust, called `Cannolib`, in order to offload some of the work that would have otherwise been done by the compiler. Moving as much complexity to Cannolib as possible results in a much simpler compiler, while simultaneously providing a more manageable code base. It is an entirely separate Rust crate that is imported from the compiled Python code. Cannolib provides all standard library functions, modules, and types that the compiled code utilizes.

Some implementations of various standard library components provide a proof of concept for others and consequently were omitted. For instance, we support Python lists but omit sets. Although Cannolib does have some type restrictions, the compiler provides a fair amount of support. While not exhaustive, types include: numbers, strings, booleans, lists, tuples, functions, and classes.

4.3.1 The Value Type

```
1  #[derive(Clone)]
2  pub enum Value {
3      Number(NumericType),
4      Str(String),
5      Bool(bool),
6      List(Rc<RefCell<ListType>>),
7      Tuple(TupleType),
8      Function(Rc<Fn(Vec<Value>, HashMap<String, Value>) -> Value>),
9      Class { tbl: HashMap<String, Value> },
10     Object { tbl: Rc<RefCell<HashMap<String, Value>>> },
11     TextIOWrapper(IOWrapper),
12     None
13 }
```

Listing 4.1: The complete Value enum encapsulating the types supported by Cannoli. This is the unoptimized version of the enum. Clone is the only derived trait, others were implemented.

To represent Python types in Rust the `enum` in Listing 4.1 was created to wrap various elements into a single construct, called the `Value` type. Python types like `Number`, `Str`, `List`, `Tuple`, `None`, etc. are all encapsulated in the `Value enum`. In combination with Rust traits, and the ability to provide functions for the `enum` data structure, this dramatically simplifies the compiled code.

An example of this would be simply adding two values. In Python the addition operator can be used to add two numbers or concatenate two strings (and may be overloaded). Rust provides an `Add` trait defined in its standard library that can be applied to a `struct` or `enum`. This `Add` trait requires the implementing type to provide an `add` method that operates on itself and another value. The logic for adding two numbers or concatenating two strings may now be done in `Cannolib` instead of generating logic to check the operands and then execute the appropriate code. This

separation of logic dramatically reduces the complexity of the compiled code while enabling the development of optimal code in the library.

4.3.2 Built-ins

Cannolib also contains Rust modules that are analogous to the built-in functions and modules in Python. The Python standard library is quite large so Cannolib only supports a subset of functions and modules. Cannoli was designed around a few benchmarks that include features analyzed in this thesis. In order to run these benchmarks, Cannolib includes `math` and `sys` modules. These modules do not completely mirror those in the Python standard library but do provide support for the aforementioned benchmarks. Along with the built-in modules are a subset of built-in functions like `print`, `len`, `open`, and `enumerate`.

Including the built-in functions and modules in the compiled code is done by importing the Cannolib modules and calling a function that returns a `Value` which is added to the current scope. For more specific Python imports, using `from-import`, Cannolib provides an object destructuring function which can be called with the names of the functions, types, or classes that need to be decoupled from the module. This architecture enables the separation of development between compilation and standard library support.

4.4 Compiling Python

As the following subsections will detail, translating Python to Rust was mostly challenging due to the enforcement of ownership rules by the borrow checker. Replicating scope and a number of Python types in Rust required the use of the dynamic borrow checker. This deferred the enforcement of ownership rules to run time. Unfortunately, optimizations that take advantage of the compile-time borrow checking rules are lost.

Aside from the ownership caveat, the translation to Rust was surprisingly straightforward. The subsequent subsections detail the more complex translations of Python elements to Rust.

4.4.1 Scope

To replicate Python's scoping rules, as described in Section 2.2.1, the compiler must output code that manages scope at run time. A single scope context is represented by a Rust `HashMap`, mapping identifiers to `Value` variants. A chain of scopes forms when functions are nested. This scope chain is represented as a Rust `Vec` of the aforementioned hash tables. When the compiler encounters an identifier it will generate a call to a function, provided by Cannolib, that searches the vector of tables for the given identifier and returns a `Value` or invokes a run time error. A call to a separate Cannolib function, that updates the scope, is generated when a value is being assigned into an identifier. Whenever a function is encountered, the current scope list is copied into the function context and a new table is appended as the local scope.

When a Python function is defined, it captures the encapsulating scope. However, changes to the encapsulating scope may still be made after the function definition. This can result in the bindings within scope changing between function calls. If a copy of the current scope list was made at the definition of a function, the currently defined scope elements would be the only encapsulated elements. Therefore, subsequent changes to the encapsulating scope would not reflect in the functions scope list.

```
1  def example():
2      try:
3          print(x + ' ', end='')
4          print(y)
5      except NameError:
6          print('404')
7  example()   # prints '404'
8  x = 'hello'
9  example()   # prints 'hello 404'
10 y = 'there'
11 example()   # prints 'hello there'
```

Listing 4.2: Unordered scope elements

Listing 4.2 demonstrates the functionality of changing the encapsulating scope. Lines 1-6 define a function, `example`, that attempts to print the identifiers `x` and `y`. A `NameError` exception is thrown if the identifers do not exist in the current scope and "404" is printed. A call to `example` is made on Line 7 but the identifiers `x` and `y` have not been defined, thus the exception is raised. Line 8 introduces `x` to the global scope, the subsequent call to `example` now locates `x` and prints its value then throws an exception when evaluating `y`. Finally, `y` is introduced and the last call to `example` on Line 11 successfully prints the values bound to `x` and `y`.

Replicating this functionality leverages references with interior mutability, discussed in Section 2.3.4. Rather than copying a scope table, a reference to a scope table is cloned. Subsequent elements added to an encapsulating scope will now be available to the function call at run time. Ultimately, the scope list is defined as a vector of pointers to shared mutable hash tables. Accessing the scope list now requires a method call, `borrow()` to obtain an immutable reference and `borrow_mut()` to obtain a mutable reference. Borrow checking the access to the scope list is therefore

deferred to run time.

4.4.2 Static Single Assignment

The borrowing rules described in Section 2.3.3 coupled with shared mutable scope (Section 4.4.1) complicate the translation to Rust. Listing 4.3 exemplifies the difficulty of directly translating Python to Rust. A direct translation, of the Python assignment on Line 2, into the scope table on Line 5 errors at run time. The borrow rules have been broken by requesting an immutable borrow of `scope_tbl` while already possessing a mutable borrow of `scope_tbl`.

```
1 // Simple assignment in Python
2 x = y + z
3
4 // Problematic direct translation to Rust with Cannoli scope
5 scope_tbl.borrow_mut().insert('x', scope_tbl.borrow().get('y').unwrap().
      clone() + scope_tbl.borrow().get('z').unwrap().clone());
```

Listing 4.3: Most direct translation to Rust, resulting in a run time error

Mitigating the borrowing conflicts, that will ultimately cause run time errors, is done by outputting Rust in static single assignment form (abbreviated SSA form). SSA form requires that there is only one assignment targeting each variable of the program [27].

```
1 // Simple assignment in Python
2 x = y + z
3
4 // Translation to Rust in SSA form
5 let v1 = scope_tbl.borrow().get('y').unwrap().clone();
6 let v2 = scope_tbl.borrow().get('z').unwrap().clone();
7 let v3 = v1 + v2;
8 scope_tbl.borrow_mut().insert('x', v3);
```

Listing 4.4: Static single assignment translation

SSA form is illustrated in Lines 5 through 8, of Listing 4.4, and is output as follows. The lexer and parser ultimately transform the statement `x = y + z` into an abstract syntax tree (AST) which the compiler will recursively traverse. An assignment statement is the first node encountered in the AST for the code on Line 2 in Listing 4.4, which includes a `target` and `value`. The `value` is a binary expression with a left and right operand and a single operator. Both the left and right operands are expressions so the compiler recurses again and finds two terminal nodes, the names `x` and `y`, which it assigns to the values `v1` and `v2`. These values are returned and the binary expression applies the operator to the returned values and assigns it into a new variable `v3`. This value is again returned to the assignment statement which simply outputs the insertion of the value into the target. In general, each expression will be assigned to a new variable, hence the single static assignment form.

```
1 fn main() {
2     println!('{}', func())
3 }
4 fn func() -> usize {
5     let v1 = 1;
6     let v2 = 2;
7     let v3 = v1 + v2;
8     v3
9 }
```

Listing 4.5: Example of constant propagation and folding

Leveraging SSA eliminates the borrow checking error resulting from conflicting borrow types. Though this may seem detrimental to performance, the Rust and LLVM compilers implement numerous optimizations, including copy and constant propagation to avoid unnecessary assignments. Listing 4.5 illustrates a toy example that benefits from constant propagation. The variables `v1`, `v2`, and `v3` are unnecessary and can be rewritten as a return statement, returning the value `3`. This is exactly what the Rust compiler does.

```
1 ; func
2 define internal i64 @func() unnamed_addr #0 {
3 start:
4     br label %bb1
5 bb1:
6     ret i64 3
7 }
```

Listing 4.6: LLVM generated by the Rust compiler

The LLVM intermediate representation generated by the Rust compiler, for the function from Listing 4.5, is shown in Listing 4.6. The return instruction on Line 6

in Listing 4.6 is a result of the compiler optimizations eliminating the unnecessary assignments. The entire body of function, `func`, was replaced with the value `3` when Listing 4.5 was compiled with the highest optimization level enabled.

4.4.3 Operators

```
impl std::ops::Add for Value {
    type Output = Value;
    fn add(self, other: Value) -> Value {
        match (self, other) {
            (Value::Number(lhs), Value::Number(rhs)) => {
                Value::Number(lhs + rhs)
            },
            (Value::Str(lhs), Value::Str(rhs)) => {
                Value::Str(lhs + &rhs)
            },
            _ => unimplemented!()
        }
    }
}
```

Listing 4.7: Implementing the Add trait for Value

Translating both unary and binary operators to Rust was pleasantly straightforward. Rust provides operator overloading through trait implementation. A struct or enum may implement a trait which is associated with an operator. For example, the `+` operator may be used for both numbers and strings in Python. Implementing the Rust trait `std::ops::Add` for the `Value` type (Section 4.3.1) is demonstrated in Listing 4.7. Implementing this trait enables the use of the `+` operator between two operands of the type `Value`. The trait implementation defines which variants of `Value` are actually supported. In Listing 4.7, `Number` and `Str` are the only supported

variants. Attempts to add unsupported `Value` variants would result in a run time error. Supporting other variants is as simple as including another pattern in the match statement. This reduces the complexity of the compiler by leveraging Cannolib to resolve operations between `Value` variants and handle errors appropriately.

4.4.4 Functions

Cannoli translated functions are designed around the way scope is managed at run time. The encapsulating scope needs to be captured when a function is defined. This functionality can be obtained by using Rust's closures (anonymous functions that capture their environment) which are exposed through three different call operator traits:

- `FnOnce` consumes variables of the enclosing scope that are used in the closure. The closure takes ownership of these variables entirely. The `Once` suffix means that the closure can not take ownership of the same variables and may therefore only be called once.
- `Fn` immutably borrows environment values
- `FnMut` mutably borrows environment values

Functions in Cannolib are defined with the `Fn` trait. The closure traits are only relevant for moving the current scope list into the functions local environment. Rust provides a `move` keyword that can be applied to a closure, which forces the closure to take ownership of the values it uses from the environment. We move the current scope list into the closure to ensure that we have the right enclosing environments.

```
1  ...
2  let move_scope = scope_list.clone();
3  let closure = || {
4      // immutable borrow of move_scope here
5      let mut scope_list = move_scope.clone();
6      // append the function's local scope table
7      scope_list.push(Rc::new(RefCell::new(HashMap::new())));
8      ...
9  };
10 ...
```

Listing 4.8: Moving the scope list into a closure

Listing 4.8 illustrates a simplified output of the Cannoli compiled code. A copy of the current scope is made in Line 2 in order to avoid losing ownership of the working scope list. This copied value is then moved into the closure and cloned into a mutable variable in order to maintain the `Fn` trait requirements. Cloning the scope list simply cloned the vector of reference counted pointers and increased the reference count of each pointer. We now have a scope list in the function that has access to its encapsulating scope.

Every Python function compiles to the same function signature in Rust. A function is a `Value` just like every other type. The function signature consists of two parameters, a vector of values and a hash table whose keys are strings and values are `Value` variants. The return value is a single `Value`, since all types are encapsulated in the `Value` enum. After the scope list is moved into the closure, an iterator is created to go through all positional arguments contained in the vector parameter and input them into the local scope table. The hash table parameter is for keyword arguments and is consumed by the local scope table. After consolidating the arguments and local scope the translation continues as normal.

The Cannolib `Value` enum implements numerous methods. The translation for invoking a function in Python is to call a method named `call()` on the `Value`, passing in the positional and keyword arguments in the appropriate format. Cannolib then handles the closure execution and propagates the return value.

4.4.5 Classes and Objects

Classes and objects are slightly different enum variants in Cannolib. Method calls are the main reason for this separation. When a class's static method is called, a reference to itself is not passed in as the first parameter. Alternatively, when a method is called on an instantiated object, a reference to itself is passed in as the first parameter, usually referred to as `self` in Python.

Classes and objects are defined with shared mutable containers, similar to scope tables. In fact, the hash table definition is the exact same as the scope table, it is just contained in the `Value` enum. Cannolib provides the functionality to access attributes and invoke methods. Attribute access, e.g. `obj.attr`, will be compiled to a method call passing off responsibility to Cannolib. Cannolib will search for the attribute name in the internal hash table and return the corresponding `Value`. Invoking class or object attributes is a similar process with the caveat of deciding to pass `self` as a parameter.

4.4.6 Modules

Modules are ultimately objects, though they are structured differently in the compiled code. All modules are output to a single file. The Python file that was initially run, whose name becomes `"__main__"`, is output into a module called `main` with a single function named `execute`. The compiler adds a module to a global queue whenever it encounters an `import` statement. Then it brings the module into scope, calls a

function named `import_module`, and inserts the result into the local scope table with the appropriate alias. Each module in the queue is compiled into its own Rust module with the `import_module` function. The `import_module` function returns the object variant of the `Value` enum.

```
1  extern crate cannolib;
2  fn main() {
3      main::execute();
4  }
5  pub mod main() {
6      pub fn execute() {
7          ...
8          use custom_mod;
9          scope_list.insert('alias', custom_mod::import_module());
10         ...
11     }
12 }
13 pub mod custom_mod {
14     use cannolib;
15     pub fn import_module() -> cannolib::Value {
16         ...
17     }
18 }
```

Listing 4.9: Cannoli module structure

Listing 4.9 presents an example of the module structure. The `main` function on Line 2 is the entry point for program execution which then calls to the `main` module. Subsequent module usage is displayed on Lines 8 and 9. A reference to built-in modules is used in the compiler to output similar code to Line 9, the module is just imported from modules within Cannolib rather than the current file.

4.4.7 Other Nuances

Other elements of note are operator chaining, short-circuiting, and assignment unpacking. Python supports comparison operator chaining, which allows expressions like `x < y and y < z` to be written as `x < y < z`. This is not functionality that Rust supports, therefore the translation expands operator chaining into a more explicit version.

```
1  // Python code checking object existence before checking its attribute
2  if obj and obj.attr:
3      ...
4
5  // Simplified translation to Rust that preserves short-circuiting
6  let v1 = cannolib::lookup_value(&scope, 'obj');
7  let v0 = if v1.to_bool() {
8      let v3 = cannolib::lookup_value(&scope, 'obj');
9      let v2 = v3.get_attr('attr');
10     if v2.to_bool() {
11         cannolib::Value::Bool(v2.to_bool())
12     } else {
13         cannolib::Value::Bool(false)
14     }
15 } else {
16     cannolib::Value::Bool(false)
17 };
18 if v0.to_bool() {
19     ...
20 }
```

Listing 4.10: Short-circuiting translation

Short-circuiting caused an issue when the switch was made to output the compiled code in SSA form. This resulted in diassembling expressions into local variables.

Translating expressions that check for existence before attempting to access an attribute, as on Line 2 in Listing 4.10, would fail because the SSA form would try to evaluate the attribute access (`obj.attr`) and assign it to a new local variable before applying the `and` operator. Rust conditionals are, conveniently, expressions which allows for their use in assignments. Short-circuiting was manually enforced and the result assigned into a local variable to satisfy SSA.

Listing 4.10 demonstrates short-circuiting translation. Line 7 checks the existence of the object located in the current scope on Line 6. The attribute is fetched and then evaluated as a boolean on Line 10. A single boolean value is then encapsulated in the `Value` type and assigned into `v0`. The result of the conditional guard on Line 2 ultimately becomes the variable `v0` that is defined on Line 7 and used on Line 18. Finally, Line 19 would continue the translation of Python elements starting at Line 3.

The Python abstract syntax tree allows the target of an assign statement to be an expression. This enables assignments into tuples and lists among other things. This is not unique to assignments and also shows up in list comprehensions and `for` statements. To correctly translate this functionality, a function was written that tail-recursively unpacks a value and assigns its contents to each target. If a list or tuple is encountered, code is output that properly indexes the value. This indexing is done through a method implemented on the `Value` enum and is thus handled by Cannolib where it may perform error checking as needed.

Chapter 5

OPTIMIZATIONS

The initial implementation of the Cannoli compiler supports an intentionally restricted subset of Python. In particular, the ability to augment an object's structure (attributes) post-creation has been eliminated. These features were eliminated in an attempt to quantify the cost to support such features in a language. This cost is an artifact of the performance improvements that cannot be implemented when these features are supported. We chose features of the language that we hypothesize are used infrequently, that could be trivially rewritten in a more static manner, or that are clearly detrimental to static analysis. Previous work attempts to quantify the use of dynamic features supported by Python [10]. They concluded that a substantial portion of Python programs use dynamic features during start up, with a considerable drop in use after. Another study reports that dynamic features are used more uniformly throughout the lifetime of Python applications [1]. Although, their experiments showed that object structure changing at run time was used less frequently than other dynamic features. The exception was adding an attribute to an object, which may be misleading since the structure of an object is typically determined by attribute assignments in the `__init__` method which may be considered uses of this feature.

Our goal is to restrict features of Python in order to show a considerable increase in performance. By doing so, we provide empirical data to language designers who may be considering a variety of features for a given language.

5.1 Restricting Dynamic Code

The first optimization concerns dynamic code. This is code that is constructed at run time from other source code. The Python functions `exec` and `eval`, along with the `del` keyword, provide this functionality. The `exec` and `eval` functions provide a way to execute Python code or evaluate an expression respectively at run time via a string argument. This argument could theoretically be provided by a command line argument or user input. Therefore, a call like `exec('x = 1')` would introduce the value `x` to the current scope. Additionally, the `del` keyword provides the functionality to remove an element from scope. In combination, collecting all scope elements at run time is impossible. Restricting this functionality enables us to alter the way in which scope is handled at run time.

The base implementation of Cannoli outputs a list of tables to represent scope. Each table provides a mapping of identifiers to their currently bound values. The list of scope tables must be traversed whenever an identifier is evaluated at run time.

With this optimization, whenever a module or function is encountered at compile-time, all assignments are gathered from the local scope.[1] A fixed-size vector replaces the shared mutable hash table representing the local scope. This provides contiguous memory that may be accessed via offsets, eliminating the overhead of a hashing function. The compiler now maintains a list of hash tables whose keys are strings representing identifier names and whose values are indices into the appropriate scope table. When an assignment or definition is encountered the compiler will output code using indices to access the appropriate scope table and the appropriate value location, rather than traversing the scope list checking each scope table.

This optimization eliminates the ability to support `exec`, `eval`, and `del` and also

[1] Making this pass in the compiler was a decision made to save time, gathering scope elements in the parser would have been a much more elegant solution.

restricts the use of variadic functions. Python supports both a variadic positional parameter as well as a variadic keyword parameter. It is seemingly impossible to statically determine the contents of these variadic parameters so this functionality falls back to hash tables representing the current scope. Finally, all scope elements must be inferable at compile time, precluding the functionality supported by `exec` and `del`, in order to observe performance increases from this optimization.

5.2 Restricting Dynamic Objects

Dynamic objects are those whose structure can be modified at run time. This includes, but is not limited to, adding attributes and methods, deleting attributes and methods, and modifying the inheritance of a class. Since Cannoli does not currently support inheritance we focus on the addition and deletion of attributes and methods. Python provides a constructor method that is called when a class is instantiated. This method is called `__init__` and takes a reference to the newly instantiated object as its first parameter, usually referred to as `self`. Within the `__init__` function programmers may add attributes by simply assigning to a new attribute associated with `self`. This is technically an example of the dynamic functionality that enables the addition of attributes at run time. Alternatively, attributes that are defined within a class are static variables. These static attributes may be overridden by instances of the class. This functionality is exemplified throughout Listing 5.1.

```
1  class Example:
2      x = 1                    # static variables
3      y = 2
4      def __init__(self, y): # constructor
5          self.y = y           # run time modification of self
6
7  a = Example(99)
8  Example.y, a.y         # (2, 99)
9  Example.x = 24         # reassigns Example.x
10 b = Example(99)
11 Example.x, a.x, b.x  # (24, 24, 24)
```

Listing 5.1: Python class attribute definitions

Disallowing dynamic objects fixes the set of attributes at creation. Thus, the object optimization works similarly to the scope optimization in that a fixed-size vector indexed by offset is much faster than a hash table indexed by arbitrary key. Cannolib can replace an object's hash table with a vector if all attributes of a class can be determined at compile-time. In order to maintain backwards compatibility, the hash table contained in the object and class `Value` variants, described in Section 4.4.5, was changed to a vector of `Value` variants with an auxiliary hash table. The auxiliary hash table maps field names to their respective indices.

The compiler determines the attributes of an object by scanning the definition for static variables and methods. If an `__init__` function is provided, the compiler will locate all attribute assignments to the parameter representing the reference to the newly instantiated object. Scanning the `__init__` function is not perfect. The `__init__` function could be replaced during run time or the `self` value could be reassigned within the `__init__` function. Both of these cases are restricted but are not currently enforced. Ideally there would be a more explicit way to define non-static

class attributes but doing so would require additions to the language.

```
1 class Example:
2     def __init__(self):
3         self.x = 'value'
4
5 obj1 : Example = Example()
6 obj2 = Example()
7
8 obj1.x # directly accesses the object vector
9 obj2.x # calls Cannolib defined method to search for the attribute
```

Listing 5.2: Annotating types for optimization

Variable assignment with annotation is shown in Line 5 of Listing 5.2. Line 6 shows the standard assignment without annotation. For our purposes, annotations are syntactically supported in function parameters and assignments. When a value is annotated the Cannoli compiler will search an internal map with all class definitions and attempt to output code that directly accesses the objects internal vector. If the compiler is unable to determine the type of a value that is currently being accessed, a method will be called to locate the value within the object's internal data structures. As previously mentioned, an auxiliary table is associated with the `Value` vector. This auxiliary hash table contains a mapping of identifiers to vector offsets and is only used if an object's type cannot be determined by the compiler. The attribute in question will be searched in the auxiliary hash table which will produce an index into the `Value` vector.

To enable this optimization, adding an attribute outside of the `__init__` function and deleting an attribute are no longer valid. Therefore, a class is defined by its class attributes and the attributes used in the `__init__` function.

Chapter 6

RESULTS

This chapter evaluates the performance of Cannoli generated code on a set of micro-benchmarks and a more substantial program.

6.1 Benchmark

Cannoli's feature set is sufficient to compile a ray casting program that has been used in introductory courses at Cal Poly. The ray caster takes an input file that specifies sphere coordinates as well as color, light, and material properties. The output is a ray traced image of the input file in portable pixmap format (PPM). Even with a simple input file, the ray caster creates, destroys, and accesses thousands of objects. It is a fantastic benchmark for testing both optimizations as they are applicable throughout this program. We compare the optimized version of Cannoli to the unoptimized version to approximate the cost of the removed language features. To satisfy curiosity, Cannoli is compared to both CPython and PyPy though there was no expectation that Cannoli would outperform these implementations. Instead, we suggest that similar performance improvements shown with Cannoli may be observed in CPython if Python were similarly restricted.

6.2 Scope Optimization

The chart in Figure 6.1 shows the average run time for the ray casting program. The first iteration of Cannoli performs significantly slower than CPython. However, restricting dynamic code and optimizing scope management, as mentioned in 5.1, provided a 40.11% increase in performance from the original Cannoli compiler. We

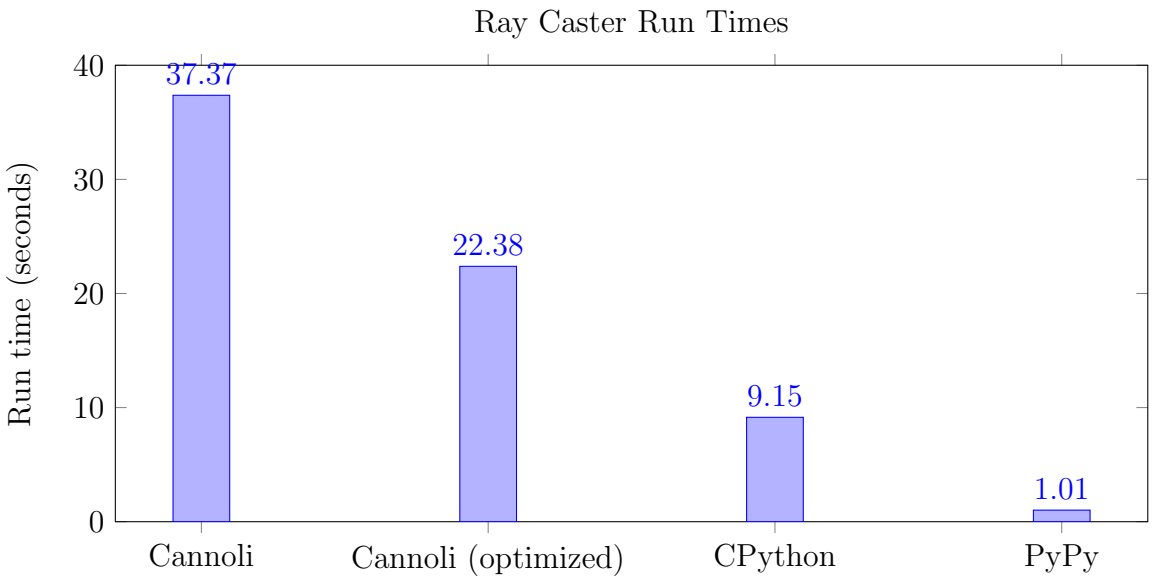

Figure 6.1: The average run time across eleven runs of the ray casting program across various Python implementations. Optimized Cannoli obtained a 40.11% speed up although it is still slower than both CPython and PyPy. The PyPy results were incredibly unexpected and outperformed CPython by a much larger margin than what was anticipated.

include the comparisons to both CPython and PyPy in order to provide a reference point for Cannoli's overall run time. Both CPython and PyPy are much more mature implementations. Due to our limited time we did not expect to outperform these implementations but, rather to illustrate performance improvements as a result of optimizations.

40.11% is a significant increase in performance considering that this is a single optimization. The original Cannoli implementation searched a vector of hash maps, from local scope to global scope, to locate a variable. Most of the variable access in the ray caster was in the local-most scope. Profiling data supports that a majority of the speed increase is coming from legitimate O(1) look up time from fixed-size vectors. Although hash table look up is theoretically O(1), the overhead of hashing still accrues a cost over time. This optimization seems to illustrate that hypothesis.

A profile was done on the compiled ray caster code (unoptimized), which ran for 42.62 seconds. Retrieving a value from the scope and object tables accounted for 6.63 seconds and 2.05 seconds respectively, a combined 8.68 seconds. Inserting elements into hash tables accounted for 4.22 seconds. Additionally, cloning hash tables accounted for 3.07 seconds. In combination, a minimum of 37.47% of the performance time is attributed to hash table complexity which correlates nicely to the 40.11% performance increase when optimized. Furthermore, a profile was done with the scope optimization enabled resulting in a total run time of 23.95 seconds. Roughly 2.5 seconds were spent on hash map functionality (which came from objects). The most significant cost of using vectors (instead of hash tables) was cloning which accounted for 1.27 seconds total.

6.2.1 What Do These Results Mean?

Python features that provide the functionality to introduce or remove an element from scope is quite damaging to run time performance. There is a significant cost attached to the functionality provided by Python's `exec` and `eval` functions, as well as the `del` keyword. Excluding dynamic code functionality from a language can provide substantial increases in performance. Though the 40.11% increase seen here is particular to the Cannoli implementation and this benchmark, it is evident that supporting these features is costly when they preclude or hinder optimizations.

6.3 Object Optimization

Performance gains were achieved when enabling the object optimization as described in Section 5.2 and illustrated in Figure 6.2. Surprisingly, the change in object data structures resulted in a performance increase even without annotations. The change from accessing a hash table of values to accessing a hash table of offsets resulted in

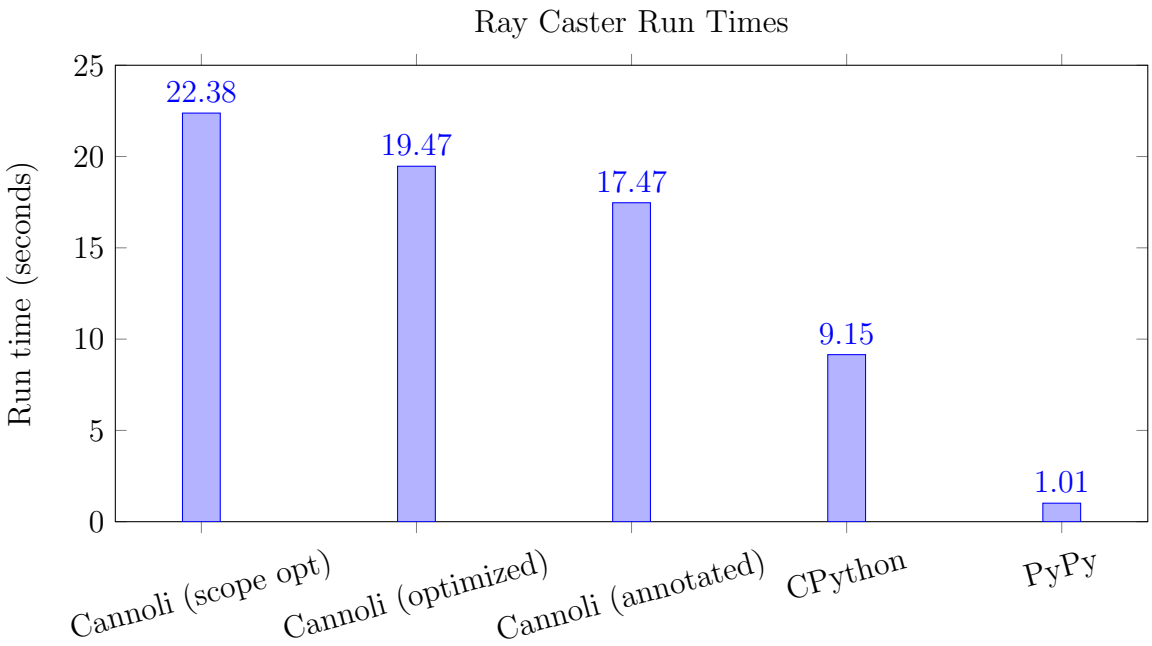

Figure 6.2: The average run time across eleven runs of the ray casting program across various Python implementations. Cannoli (scope opt) is a version of Cannoli with just scope optimization enabled. The Cannoli (optimized) bar is the previous scope optimization paired with the new object-vector model. The Cannoli (annotated) bar is an annotated version of the ray caster that enables the object optimization. We again include CPython and PyPy as a reference point to the Cannoli run times.

performance improvements alone. We again speculate that this is due to overhead associated with hash tables and the ability to access contiguous memory via the fixed-size vector. In order to obtain compiled code that was directly accessing the object data structures, instead of calling to Cannolib, the ray caster had to be annotated. Every variable and function parameter that was bound to an object type was annotated with the corresponding class. Enforcing static object definitions and annotating the ray caster allowed us to obtain a 22% speed up from the scope optimization. The Cannoli (annotated) bar in Figure 6.2 is our fastest version of Cannoli. It combines both optimizations with annotations to produce a 54% increase in performance from the original Cannoli compiler.

Input	Cannoli	Cannoli (optimized)	CPython	PyPy
Bunny (small)	15m21.776s	6m11.417s	5m1.485s	0m2.914s
Dragon	182m37.962s	79m0.474s	61m16.823s	0m42.683s
Bunny (large)	283m38.243s	122m57.557s	184m30.911s	1m37.133s

Table 6.1: Larger inputs for the ray casting benchmark. The inputs are ordered from smallest (top) to largest (bottom). Cannoli (optimized) begins to outperform CPython on the largest input.

6.3.1 What Do These Results Mean?

Removing the ability to introduce or remove attributes and methods from a class at run time results in improvements to performance. Improvements were seen by just enforcing static object definitions which allowed us to better control the physical layout of memory by creating fixed-size vectors. Further improvements were achieved by annotating assignments and function parameters with an object's associated class. This allowed the compiler to directly access the values of an object without searching for them at run time. Static object definitions provide more information to the compiler that is used to improve attribute use at run time, leading to a 22% increase in Cannoli performance from the previous optimization.

6.4 Larger Inputs

The ray caster was run with three much larger inputs, the results are presented in Table 6.1. Each input constructed an image from ray-traced spheres. Profiling data from the original input showed that Rust's file I/O was responsible for an unusually large amount of the total run time (this is further covered in Section 6.5.1). The largest input amortizes the file I/O slowdown and shows that, at scale, the optimized version of Cannoli performs the ray casting computations much faster than CPython. Again, PyPy's performance is significantly faster than both Cannoli and CPython.

Test	Description	Label
1	26 levels of nested scope elements accessed 1 million times	scope
2	calling empty function 1 million times	empty func
3	incrementing local variable 10 million times	inc var
4	print local variable 10 million times	print var
5	print local variable 10 million times w/o newline	print var 2
6	pushing and popping 10 million list elements	push-pop
7	cloning list via slices 10 million times	slice clone
8	creating 10 million objects	create obj
9	calling object's method 10 million times	methods
10	1 million list comprehensions on a list of 100 elements	listcomp
11	reversing list of 100 elements 1 million times via slices	reverse list
12	alternating conditional branching 10 million times	branch swap
13	indexing list 10 million times	index list
14	accessing object 10 million times	obj access
15	accessing annotated object 10 million times	ann access

Table 6.2: Descriptions of each benchmark.

6.5 Microbenchmarks

Fifteen benchmarks were designed to test various elements of Python across four different implementations. These four implementations are Cannoli with no optimizations enabled, Cannoli with all optimizations enabled, CPython, and PyPy. Descriptions of each microbenchmark are listed in Table 6.2. Table 6.3 presents an average of 10 runs for each benchmark per implementation and shows that Cannoli (optimized) outperforms CPython in six of the 15 benchmarks. PyPy is significantly faster than the other three implementations while Cannoli (unoptimized) is the slowest. Further-

Test	Label	Cannoli	Cannoli (opt)	Improvement	CPython	PyPy
1	scope	9.440	1.434	84.808%	**0.270**	0.087
2	empty func	0.379	**0.210**	44.647%	0.254	0.076
3	inc var	1.675	**0.468**	72.079%	1.184	0.083
4	print var	18.515	16.279	12.077%	**6.603**	3.097
5	print var 2	8.424	**5.868**	30.347%	9.254	5.495
6	push-pop	4.194	**1.994**	52.468%	3.585	0.643
7	slice clone	24.260	20.657	14.854%	**2.696**	0.492
8	create obj	15.636	7.338	53.071%	**5.265**	0.175
9	methods	9.928	5.077	48.864%	**3.432**	0.105
10	listcomp	13.816	4.938	64.259%	**3.190**	0.608
11	reverse list	4.653	3.999	14.064%	**0.681**	0.225
12	branch swap	2.158	**0.768**	64.396%	1.711	0.091
13	index list	6.510	**1.029**	84.195%	1.565	0.081
14	obj access	6.510	3.249	50.087%	**2.616**	0.085
15	ann access	6.511	2.917	55.201%	**2.644**	0.083

Table 6.3: The average run times (in seconds) of 15 benchmarks across four Python implementations. An average was taken of 10 runs per implementation. PyPy is the fastest implementation across all benchmarks, base Cannoli is generally the slowest. Cannoli (optimized) and CPython are the focus of this table, the faster implementation between the two is bolded for each benchmark. The improvement of Cannoli (optimized) over base Cannoli is also presented.

more, the performance improvements of the optimized version of Cannoli over the unoptimized version is also presented in Table 6.3. The optimized version of Cannoli achieves an average performance improvement of 50% across all 15 microbenchmarks, which correlates to the 54% achieved for the ray casting benchmark.

6.5.1 Outliers

The optimized version of Cannoli performed fairly well against CPython on average. However, there are three benchmarks that stand out as notable outliers. Benchmarks 4, 7, and 11 run considerably slower than CPython and also benefit from optimizations significantly less than the other benchmarks.

Benchmark 4 simply prints a local variable containing the string "hello" to `stdout` ten million times. Table 6.3 shows that optimized Cannoli runs this program in 16.279 seconds on average while CPython runs it in 6.603 seconds on average. Profiling data shows that 70.6% of the execution time was spent in Rust's standard library on calls that write to `stdout`. Cannoli uses the macros `print!` and `println!`, which Rust provides as the primary form of output to `stdout`, and both are line-buffered [15]. As a result, the main slowdown comes from 51.5% of the execution time being spent on calls to an internal `flush` method that flushes the buffer for `stdout`. Avoiding line buffering, as is done in benchmark 5, results in a significant improvement in performance. The amortized overhead observed in the ray caster with larger inputs is attributed to these I/O oddities.

Both benchmarks 7 and 11 concern list manipulation via Python slices. Cannoli's implementation of slices takes a functional approach rather than CPython's imperative approach [29]. This results in the use of iterators that apply various functions to a list's elements to return the appropriate slice. Alternatively, CPython uses pointers to construct slices from lists [29]. After profiling both benchmarks 7 and 11, 85.6% and 89.8% of the execution time, respectively, was spent in Cannolib's `slice` method. Benchmark 7 spent a significant amount of time calling a `next` method internal to Rust iterators, while benchmark 11 spent time calling `map` and `reverse` methods. It should be noted that Rust does provide slicing as a feature; however, it differs significantly from Python's and, thus, cannot be used as a direct replacement.

Chapter 7

FUTURE WORK

A significant portion of the work in this thesis was developing a compiler that supports a reasonable subset of the Python language. Dynamic code and objects were then evaluated when Cannoli was stable enough to perform an analysis on these features. There are a few areas for future work that should be considered.

Work can be done on the Cannoli compiler to support Python features that were omitted solely in the interest of time. These include exceptions and inheritance as well as a more complete standard library implementation. As mentioned before, Cannoli's feature set was developed to the point of successfully compiling the ray casting benchmark. Therefore, numerous proofs-of-concept were implemented instead of developing an exhaustive feature set. For example, dictionaries and sets were not implemented but their implementations would follow the same structure as the `list` implementation. Other type support like associated methods or built-in functions could also be extended. Examples of this include adding list methods like `remove`, `reverse`, and `sort` or built-in functions `type`, `min`, and `max`.

As far as evaluation is concerned, other dynamic features should be considered for analysis. We covered two major dynamic features in this thesis, scope modification and dynamic objects. Exploring restrictions on types and leveraging type annotations would be another interesting analysis. For instance, the ray casting benchmark would benefit substantially from type annotations on primitive types. This would eliminate the call to implemented trait methods, as described in Section 4.4.3 and illustrated in Listing 4.7, allowing unboxing to be done inline.

Other studies on different subsets of Python could also be done. Table 6.3 shows

the impressive performance of PyPy's implementation of Python. As mentioned in Section 3.3, the PyPy interpreter is written in a restricted subset of Python called RPython. It then performs just-in-time compilation on the RPython code. Developing a compiler that compiles RPython ahead-of-time (AOT) could answer questions regarding the performance of AOT compilation versus PyPy's JIT compilation.

Finally, further improvements may be made to the compiled code to leverage features of Rust that may not have been utilized. Analysis may be done on various constructs pertaining to Rust that could possibly provide performance increase (e.g. the performance of structs versus enums). Exploring the "unsafe" Rust functionality could benefit both the compiled code and standard library, specifically slices which perform poorly in Cannoli (as described in Section 6.5.1). Additionally, leveraging Rust lifetimes to avoid deferring ownership enforcement to run time (as was done in Section 4.4.1) may provide performance increases. This would eliminate some cloning of values where a reference might be passed.

Chapter 8

CONCLUSION

In this thesis we present a new implementation of Python called Cannoli. Cannoli supports a subset of Python 3.6.5 and is developed in Rust while also compiling Python to Rust. An evaluation was done on two dynamic features of Python by restricting the language and applying optimizations. We show that restricting the use of dynamic code and dynamic objects provide a considerable increase in performance from optimizations that exploit the increased static information. Cannoli demonstrated a 40.11% increase in performance when restricting dynamic code and a 22% increase, from the last optimization, when restricting dynamic objects for the ray casting benchmark. The ray casting benchmark achieved a total speedup of 54% when comparing a restricted version of Cannoli to its standard implementation. This performance increase was further supported by an average speedup of 50% across 15 microbenchmarks. We suggest that similar performance improvements could be observed in CPython with similar restrictions by enabling further optimizations. The performance cost of these features have been quantified across numerous benchmarks and are available to language designers who may be considering these features for current and future languages.

BIBLIOGRAPHY

[1] B. Akerblom, J. Stendahl, M. Tumlin, and T. Wrigstad. Tracing dynamic features in python programs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 292–295, New York, NY, USA, 2014. ACM.

[2] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. Rpython: A step towards reconciling dynamically and statically typed oo languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, pages 53–64, New York, NY, USA, 2007. ACM.

[3] C. F. Bolz, A. Cuni, M. Fijalkowski, M. Leuschel, S. Pedroni, and A. Rigo. Runtime feedback in a meta-tracing jit for efficient dynamic languages. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '11, pages 9:1–9:8, New York, NY, USA, 2011. ACM.

[4] B. Cannon. Localized type inference of atomic types in python. Master's thesis, California Polytechnic State University, San Luis Obispo, June 2005.

[5] L. Dalcin, R. Bradshaw, K. Smith, C. Citro, S. Behnel, and D. S. Seljebotn. Cython: The best of both worlds. *Computing in Science & Engineering*, 13:31–39, 09 2010.

[6] M. Dufour. Shed skin an optimizing python-to-c++ compiler. Master's thesis, Delft University of Technology, April 2006.

[7] S. Guelton, P. Brunet, M. Amini, A. Merlini, X. Corbillon, and A. Raynaud.

Pythran: Enabling static optimization of scientific python programs. *Computational Science & Discovery*, 8(1):014001, 2015.

[8] J. Hallen and C. Tismer. Complete python implementation running on top of cpython. Technical report, PyPy, December 2005.

[9] G. Hoare. Rust. `https://www.rust-lang.org/en-US/`.

[10] A. Holkner and J. Harland. Evaluating the dynamic behaviour of python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ACSC '09, pages 19–28, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[11] J. Hugunin. Ironpython. `http://ironpython.net/`.

[12] J. Hugunin. Jython. `http://www.jython.org/`.

[13] J. Hugunin. Python and java: The best of both worlds. In *in Proceedings of the 6th International Python Conference. CNRI*, pages 1997–10, 1997.

[14] J. Hugunin. Ironpython: A fast python implementation for .net and mono. In *PyCON 2004*, 2004.

[15] S. Klabnik, G. Gomez, and C. Farwell. *Macro std::print*. The Rust Programming Language, 1.25.0 edition, March 2018.

[16] S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, June 2018.

[17] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[18] A. D. Mascio and L. Aubry. Report about the parser and bytecode compiler implementation. Technical report, PyPy, December 2005.

[19] E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages, 2004.

[20] S. Nanz and C. A. Furia. A comparative study of programming languages in rosetta code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 778–788, May 2015.

[21] L. D. Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2):12–15, Feb 2007.

[22] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, Oct 2000.

[23] A. Rigo. Pypy. `https://pypy.org`.

[24] A. Rigo. Psyco, the python specializing compiler, December 2001.

[25] A. Rigo, M. Hudson, and S. Pedroni. Compiling dynamic language implementations. Technical report, PyPy, December 2005.

[26] J. Roquet and T. Spura. Shed skin – an experimental (restricted-python)-to-c++ compiler. https://shedskin.github.io/.

[27] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM.

[28] TIOBE. Programming language index. https://www.tiobe.com/tiobe-index/, April 2018.

[29] G. van Rossum and R. Hettinger. Cpython slice implementation. https://github.com/python/cpython/blob/master/Objects/listobject.c#L2746.

[30] C. Winter and T. Lownds. Pep 3107 – function annotations, December 2006.