C++ to Python Language Transpiler

Ziyuan Wang

John Yu

December 11, 2023

1 Introduction

In the landscape of software development, the need for interoperability between different programming languages is paramount. C++ and Python are two of the most widely used programming languages, each with its unique strengths. C++ is known for its high performance and control over system resources, while Python is celebrated for its simplicity and extensive library support. In this environment, software reuse emerges as a crucial practice, allowing for the efficient leveraging of existing codebases to save time, reduce costs, and maintain consistency across platforms. This paper introduces a transpiler that allows developers to repurpose C++ code within Python projects without the overhead of rewriting or maintaining parallel codebases with an innovative approach centered around the concept of an intermediate representation (IR).

IR is a universal representation method, you can think of it as a bridge between C++ and Python. This representation is crucial as it abstracts the syntactic and semantic structure of the source code, facilitating optimizations and ensuring that the translated Python code mirrors the intent and efficiency of the original C++ code.

In subsequent sections, we'll explore the design, challenges, implementation, and testing of a C++ to Python language transpiler with an integrated code optimization module.

2 Design and Implementation

This section outlines the architecture and functionality of the C++ to Python transpiler. Central to its design is a modular approach, comprising the Intermediate Representation (IR), Token & Parser, Python Code Generator, and the Optimization Module. Each module plays a crucial role in the transpilation process: from analyzing and abstracting the C++ code to generating optimized Python code. We will delve into the specifics of each module, highlighting their roles and the techniques employed to ensure effective translation and optimization.

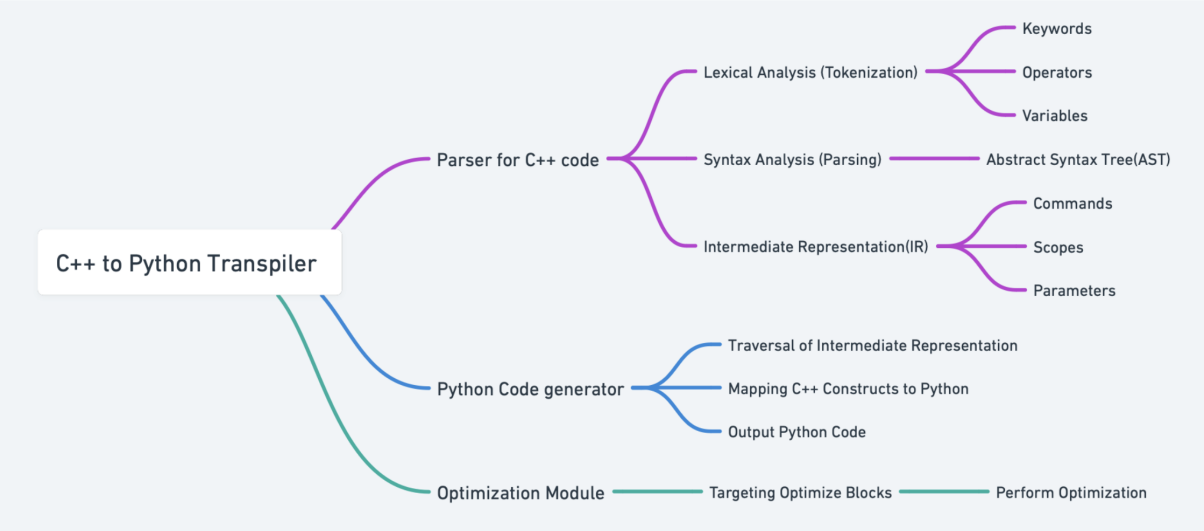

Figure 1: Structure Diagram

2.1 Intermediate Representation (IR)

The Intermediate Representation (IR) is a crucial abstraction within the transpiler architecture, serving as the essential intermediary that captures the semantics and attributes of the original C++ code. This abstraction is designed to ensure a comprehensive and accurate representation of the C++ code, which is pivotal for the translation process into Python.

The IR is structured as a series of tuples, with each tuple corresponding to a single line of C++ code. In the original idea, the first element of each tuple is the command, indicating the primary operation or execution target. Subsequent elements in the tuple

consist of parameters that carry all necessary information for the command's execution.

For example:

- A variable assignment in C++, such as `x = 11;`, is represented in the IR as `('assign', 'x', '11')`. Here, `'assign'` is the command, `'x'` is the variable, and `'11'` is the value.
- A while loop like `while (j == 5)` is translated into `('while', 'j == 5')`, where `'while'` denotes the loop command and `'j == 5'` is the condition.
- Cout statements are rendered with consideration of their arguments. For instance, `cout << "Hello World";` is transformed into `('cout', '"Hello World"')`. A compound statement such as `cout << "Hello World" << x;` is represented as `('cout', {'"Hello World", 'x'})`, note the second part will be a set of elements so it can handle multiple parameters in cout.

2.1.1 Difference in structural and syntactic

As we go deeper, we notice some differences between C++ and Python in structural and syntactic. Another important aspect to consider with IR is the difference in the way scopes and blocks are handled between C++ and Python. C++ uses braces { } to define the beginning and end of a block of code, allowing complex nested structures to be used within a single line. Instead, Python uses indentation levels to delineate blocks, making the explicit block structure mandatory for every new scope.

To address this fundamental difference, we add an additional element, termed 'scope level' or 'level of scopes' into the IR's tuple structure. Positioned as the second element in each tuple, this component captures the scope level of each line of code, thereby mapping the hierarchical context of the commands in the original C++ source code.

Consider the following C++ code, which is written in a compact, single-line format using braces:

```
if (check > 0){check += 1; if (check > 1){check += 2;} else {check += 3;}}
```

Translating this to Python requires an understanding of the nested scopes, which Python explicitly represents through indentation:

```
if check > 0:
    check += 1
    if check > 1:
        check += 2
    else:
        check += 3
```

In the IR, each tuple captures not only the command and its parameters but also the indentation depth or scope level. This information is vital for ensuring that the generated Python code accurately reflects the logical structure and flow of the C++ code, adhering to Python's indentation-based scoping.

Thus, this extended IR structure serves to represent the C++ code's semantics while embedding syntactic information crucial for translating into Python's block-structured format. This addition underscores the transpiler's sophistication in handling not just syntax but also the nuanced aspects of language-specific programming paradigms.

2.2 Token & Parser

The "Token & Parser" module of our transpiler forms the foundation of the translation process, dealing with the complexities of C++ syntax to ensure a smooth conversion to Python. This section describes how the source code is tokenized and parsed using Python's PLY library, Lex-Yacc.

2.2.1 Tokenization

The lexer breaks down the source code into tokens that represent the smallest units of meaning, such as numerals (`NUM`), variables (`VAR`), operators (`PLUS`, `MINUS`, `DIV`), and control structures (`IF`, `ELSE`, `WHILE`, etc.). Also include those special tokens for C++ specific constructs like `cout` and `endl`.

This lexer is tailored to manage C++'s syntax complexities, distinguishing between various data types and identifiers, and parsing diverse C++ statements comprising multiple operators and control structures. It includes functionality to ignore non-essential content like comments and namespace declarations, as shown in the `t_IGNORE_CONTENT` function.

2.2.2 Parsing and AST Construction

The parser processes the tokens to build an Abstract Syntax Tree (AST), which is essential for understanding the hierarchical structure of the code. The parser functions (`p_ prefixed functions`) define how sequences of tokens are combined to form valid syntactic constructs.

For example, the production rule handling function declarations (`p_func_decl`), variable assignments (`p_statement_assign`), and control structures like if statements (`p_statement_if`) and loops (`p_statement_for`, `p_statement_while`). These parsing rules are instrumental in creating an AST that mirrors the logic and structure of the original C++ code.

The parser also deals with C++'s block structure which mentioned in previous section, utilizing a SymbolTable class to manage scope levels. This is particularly important for translating C++'s brace-enclosed blocks to Python's indentation-based scope definition.

2.3 Python Code Generator

The task of Python Code Generator is converting the intermediate representation (IR) of C++ code into Python. This module is essential for ensuring that the semantics of the original C++ code are faithfully translated into Python syntax.

Translation Management: To streamline the translation of IR into Python code, a dedicated `switch` function has been implemented. This function interprets each command type within the IR and subsequently invokes the appropriate translation function.

Translation Functions:

1. **Variable Declaration (`python_var_declare`):** Converts C++ variable declara-

tions with different data type to Python format, adapting to Python's dynamic typing system.

2. **Function Definition (`python_function`):** Translates C++ function definitions into Python, accounting for differences in function signatures and argument handling.

3. **Print Statement (`python_print`):** Converts C++ `cout` statements to Python's `print` function, including the handling of string literals and newline characters.

4. **Loop Translation:** Translates C++ loops into Python's loop structures. The `python_for` function adjusts C++ loop conditions for Python's `for...in range...` format, while `python_while` preserves the condition in while loops.

2.4 Optimization Module

To give our transpiler more depth, we implemented a module for optimizing the code during the intermediate representation stage before the target language's code is generated. Given the time constraints of our project we focused on completing one optimization, that being loop reduction.

```
for (int i = 1; i < SIZE; i++) {
    a[0] = REDUCE(a[0], a[i]);
}
```

This is a variant of loop unrolling that we learned in class that works by detecting code that follows the form show in the code snippet above. If this form is detected and the REDUCE operator is associative, that means loop reduction is possible. This means that each iteration of the loop can run independently of each other. To accomplish loop reduction, we divide the input array into equally-sized partitions and add code to run the REDUCE operation for each element in all the partitions. This allows the processor pipeline to run these instructions in parallel which reduces the total execution time of the code.

```
for (int i = 1; i < SIZE/2; i++) {
    a[0] = REDUCE(a[0], a[i]);
    a[SIZE/2] = REDUCE(a[SIZE/2], a[(SIZE/2) + i]);
}
a[0] = REDUCE(a[0], a[SIZE/2])
```

After the operations in the partitions finish, we combine the partitions into the original partition by running the REDUCE operator on the first element of each partition. In order to run this optimization without user input, we created a hard-coded number of partitions. We created four partitions, which should generate a significant performance boost in languages that support parallel execution of independent computations. This type of optimization also assumes that the target array's size is evenly divisible by the number of partitions so for testing purposes we ensured that this condition was met. But under normal circumstances, a safety check is used. And if the size of the array is not divisible by the number of partitions, we also run the reduction operations on any stray elements at the end of the array that don't fit into the number of partitions.

3 Testing and Result

This section evaluates our transpiler from the two aspects of code generator and optimization, testing the accuracy of code translation and the performance of the optimization module respectively.

3.1 Tests for Code generator

In this subsection, we thoroughly examine the transpiler's ability to accurately convert C++ code into Python. This evaluation involves several test cases, each designed to target different aspects of the C++ language and its translation to Python. Key areas of focus include the handling of basic syntax, complex control structures, data types, loops, conditional statements, and function definitions and calls. For instance, a test involving nested loops assesses how the transpiler manages multiple looping constructs, while an-

other test involving function calls and array manipulations evaluates the translation of more complex interactions. Each test begins with a specific C++ code snippet, which is then translated by the transpiler. The generated Python code is executed, and its output is compared to the output of the original C++ code to ensure consistency and accuracy.

3.1.1 Test 1: Nested Loops and Conditional Statements

test_1.cpp > ...

```cpp
// Nested Loops and Conditional Statements

#include <iostream>
using namespace std;

int main() {
    int sum = 0;
    for (int i = 1; i <= 5; i++) {
        for (int j = 1; j <= i; j++) {
            if (i == j) {
                sum += i;
            }
        }
    }
    cout << "Sum = " << sum << endl;
}
```

(a) C++ Code for Test 1

python_code.py > ...

```python
sum = 0
for i in range(1, 6):
    for j in range(1, i + 1):
        if i == j:
            sum += i
print("Sum =", sum)
```

(b) generated Python Code for Test 1

Figure 2

Result Comparison: Both the C++ and Python codes output the same result: *Sum = 15*.

3.1.2 Test 2: Function Calls and Array Calculation

test_2.cpp > ...

```cpp
// Function Calls and Array coculation
#include <iostream>
using namespace std;

void incrementArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i]++;
    }
}

int main() {
    int myArray[5] = {1, 2, 3, 4, 5};
    incrementArray(myArray, 5);
    for (int i = 0; i < 5; i++) {
        cout << myArray[i] << " ";
    }
    cout << endl;
}
```

(a) C++ Code for Test 2

python_code.py > ...

```python
def incrementArray(arr, size):
    for i in range(size):
        arr[i] += 1

myArray = [1, 2, 3, 4, 5]
incrementArray(myArray, 5)
for i in myArray:
    print(i, end=" ")
print()
```

(b) generated Python Code for Test 2

Figure 3

Result Comparison: The outputs from both codes are identical: {2 3 4 5 6}.

3.1.3 Test 4: Complex Control Flow

test_3.cpp > ...

```cpp
// Complex Control Flow
#include <iostream>
using namespace std;

int main() {
    int x = 5;
    if (x > 0) {
        x += 10;
        if (x < 20) {
            x -= 4;
        }
    } else {
        x = 1;
    }
    cout << "x = " << x << endl;
}
```

(a) C++ Code for Test 3

python_code.py > ...

```python
x = 5
if x > 0:
    x += 10
    if x < 20:
        x -= 4
else:
    x = 1
print("x =", x)
```

(b) generated Python Code for Test 3

Figure 4

Result Comparison: Both the original C++ code and the Python translation produced $x = 11$.

Conclusion: The consistent results across these tests validate the transpiler's ability to accurately translate intricate C++ constructs into Python, preserving both syntax and semantics.

3.2 Tests for Optimization Module

In a compiled language such as C++, optimizations such as loop reduction via partitioning make a difference for a variety of reasons. That includes factors such as fewer branch predictions being needed and the code pipeline being able to run instructions in parallel. This is not necessarily the case in Python, which is interpreted and therefore runs in real-time. In order to test whether our optimizations made a difference in the Python code, we wrote C++ code containing optimizable loops.

We then had our transpiler transpile both an unoptimized version of the code and an

```cpp
void reference_reduction(int *b, int size) {
  for (int i = 1; i < size; i++) {
    b[0] += b[i];
  }
}

void main(){
  int size = 4444; //a multiple of 4
  int arr[size];
  for(int i = 0; i < size; i++){
    arr[i] = i;
  }

  reference_reduction(arr, size);
}
```

(a) original Cpp code

```python
def reference_reduction(b, size):
  a = [0, 0, 0, 0]
  for i in range(1, size, 1):
    a[0] += b[i]
  print(a[0])

if __name__ == '__main__':
  size = 444444
  b = []
  for i in range(0, size, 1):
    b.append(i)
  reference_reduction(b, size)
```

(b) Unoptimized python code

Figure 5

```python
def reference_reduction(b, size):
  a = [0, 0, 0, 0]
  for i in range(1, int(size/4), 1):
    a[0] += b[0*int(size/4)+i]
    a[1] += b[1*int(size/4)+i]
    a[2] += b[2*int(size/4)+i]
    a[3] += b[3*int(size/4)+i]
  a[0] += a[1]
  a[0] += a[2]
  a[0] += a[3]
  print(a[0])

if __name__ == '__main__':
  size = 444444
  b = []
  for i in range(0, size, 1):
    b.append(i)
  reference_reduction(b, size)
```

Figure 6: Optimized Code

optimized version of the code. We then ran each version of the code several times and measured how long it took each time using the time command in the terminal. Our results did not initially suggest a statistically significant improvement in performance from using loop reduction in our generated Python code. The code shown in figures 5 and 6 both take up about 0.063 of user and system time, which is the amount of time the Python code ran for.

However, after further testing, we came to the realization that this initial lack of results could stem from the total program time being dominated by the amount of time taken to fill the array. So we generated code once again, but this time the reference_reduction function was called several times. The code in both was changed to use an array size of

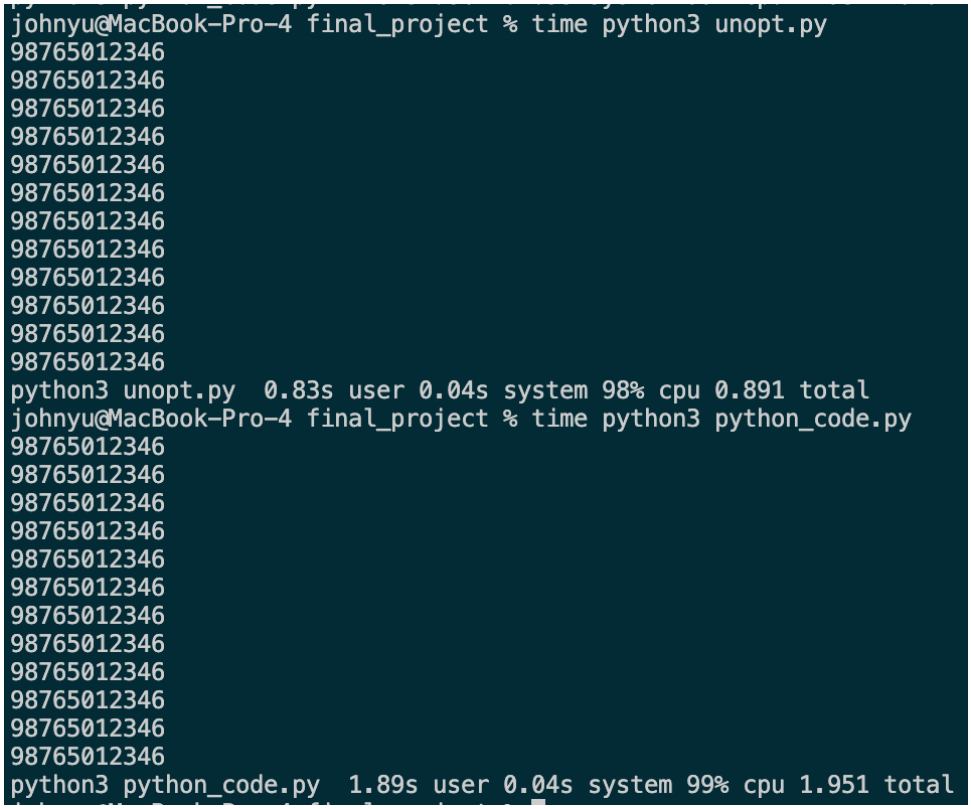

Figure 7: Result of Optimization Timing

444444 and the loop reduction section of the code was called 11 times in each program. After making this change, we got some surprising results.

The unoptimized Python code yielded faster execution times than the optimized code. The unoptimized code had results of 0.83s user 0.04s system, and the optimized code had results of 1.89s user 0.04s system. The system time, which is how long it actually takes to run the code, is the same between them. But the user time actually increases for the optimized version. This leads us to the conclusion that since Python is not compiled, it cannot take advantage of the benefits of loop reduction and unrolling the loop simply increases the number of instructions it must read which slows it down.

4 Contributions and Future Work

In this project, Ziyuan was primarily responsible for the overall design and structure of the project, including the creation of the intermediate Representation(IR), which is vital for maintaining the semantic integrity during code translation. Additionally, Ziyuan im-

plemented part of the parser functions and the translation management in Python code generator. Ziyuan also did the testing phase for the Python code generator to ensure its accuracy.

John's contribution to this project included collaborating with Ziyuan to make design decisions on the scope of our project. Along with that John also implemented part of the parser including rules for parsing declarations, functions, and if-else blocks. Furthermore, John worked on capturing potential areas in the code for optimization and writing code to generate the optimized representation then testing it.

In the future, some additions we want to make to this project are support for parsing a larger subset of the C++ language. Currently we support a fairly barebones version of C++, that could be expanded upon with more supported data types, data structures, error handling and other language features. Furthermore, we would expand the number of optimizations we implement. Right now we support one optimization, which is loop reduction. Some other ones we could add are local value numbering and interlaced loop unrolling which are topics we explored in the homework in this class. Although loop reduction did not have an optimizing effect on the Python code in practice, an optimization such as local value numbering should due to copying data being less computationally intensive than recomputing it. A more complete transpiler would also translate C++ into more idiomatic Python code, meaning Python code that takes better advantage of Python language features. These features include list comprehensions, library functions, and built-in data structures.