Grammar Based C/C++ to Rust Transpiler

Team

1. College Professor(s): Dr. Srinivas Pinisetty
2. Students:
    1. Mithun Chandrashekar
    2. Arnav Kumar Behera
    3. Vedanta Mohapatra
3. Department: Computer Science and Engineering

Date: 19 Dec 2023

RUST | Grammar based C/C++ to Rust Auto Transpiler

SAMSUNG

Problem Statement

- Rust delivers plethora of promises on memory, concurrency safety. More and more organizations are eager to move their code base to Rust.
- But converting huge C++ codebases to rust is challenging and time consuming task which can be automated with the help of state of the art transpiler
- Manually code migration also poses risk of human errors
- Requires developer to be proficient in C/C++ as well as Rust to produce efficient rust code.

Ajaganna Bandeppa
ajju.b@samsung.com
+91 9844992221

Shinde Arjun Shivaji
shinde.arjun@samsung.com
+91 9766993329

Work-let expected duration ~4 months

Expectations

- Deterministic Source to Source transpilation using grammar based approach
- Plug and play architecture to support to cope with rust lang development speed

3
Members

Training/ Pre-requisites

- Good knowledge of Grammar, Compilers
- Well versed with Rust
- Well versed with C/C++

Aug — Kick Off < 1st Month >

- Understanding of grammars, Compilers.
- Transpilation of basic C/C++ constructs
- This iteration will focus on declarations, arithmetic operations, conditionals and loops etc

Sep — Milestone 1 < 2nd Month >

- Advanced C/C++ construts transpilation
- This iteration will focus on transpiling C++ functions and struts

Oct — Milestone 2 < 3rd Month >

- Handling the concepts of ownership and borrowing in the basic constructs
- Applying the same in Advanced constructs

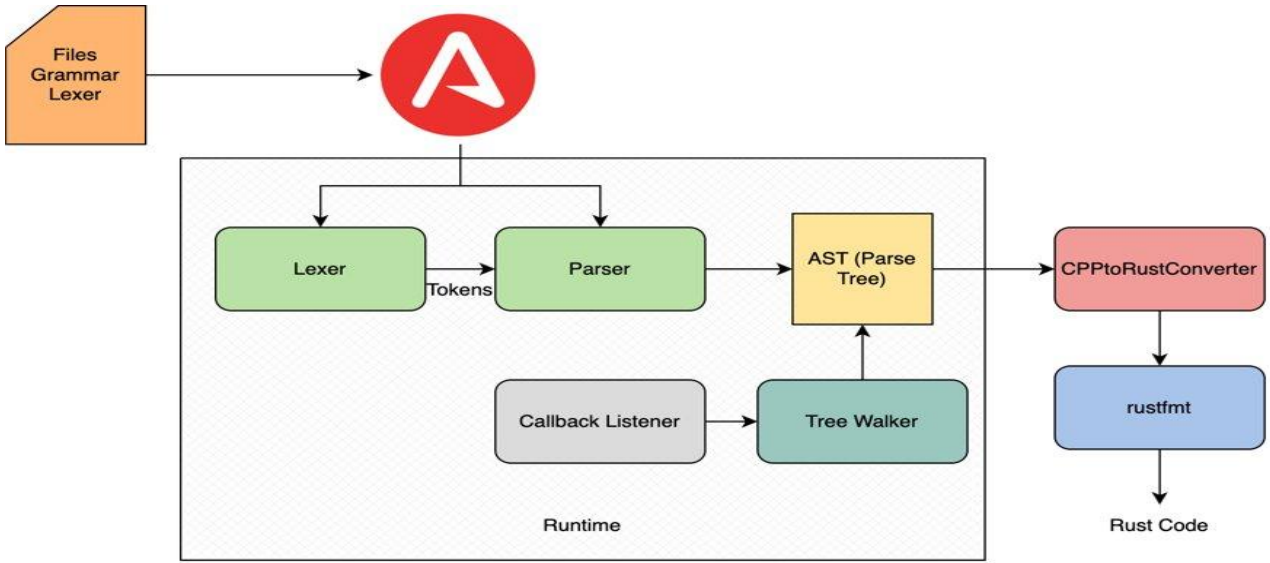
Nov — Closure < 4th Month >

- Testing
- Validation
- Optimization
- Documentation
- Deployment

Proposed Approach

We use a ANTLR generated parser. We derive a parser from the Base parser and override the visitor methods corresponding to non-terminal symbols in the grammar.

- **Concept Diagram** :



Proposed Approach

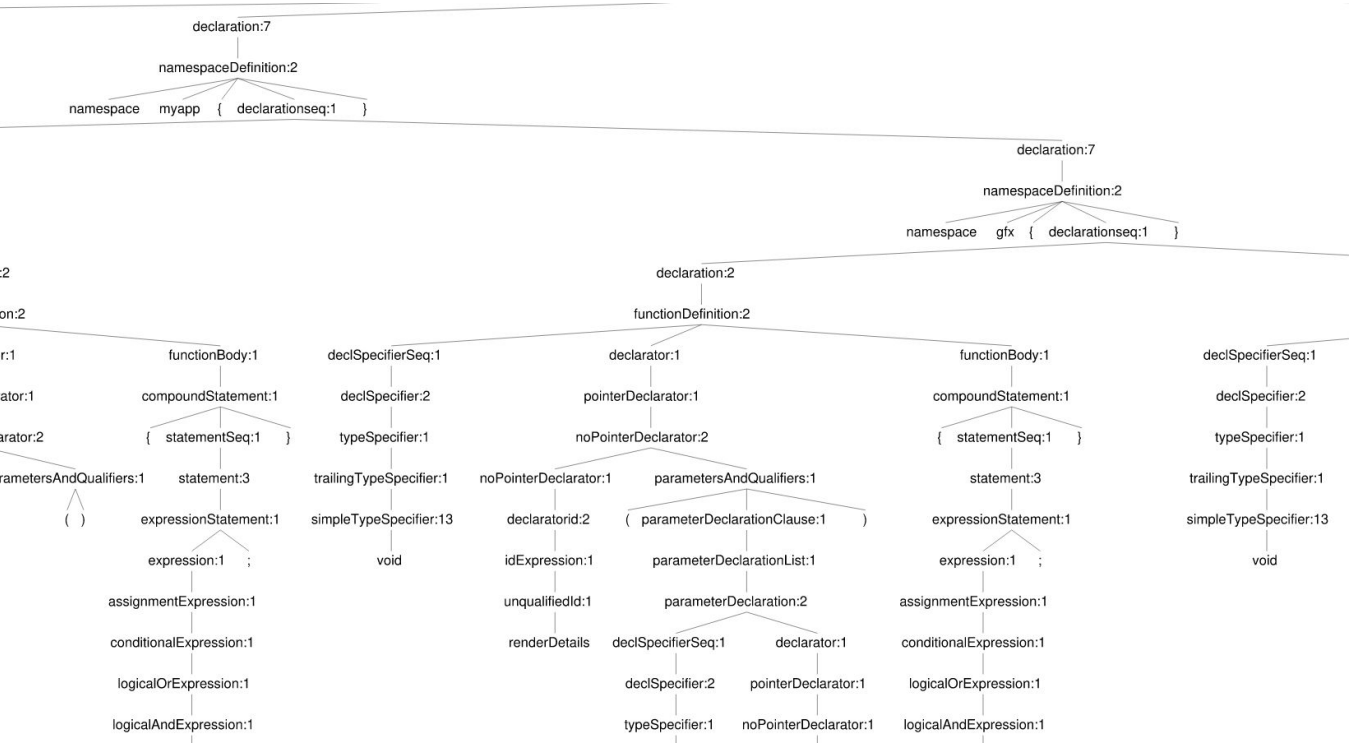
The general approach we take is to study the parse tree of C programs and look for the subtrees corresponding to the constructs we wish to transpile and override corresponding visitor methods

- **A Sample Parse Tree for Simple Declarations** :

- declSpecifierSeq , initDeclaratorList are the relevant non-terminals.
- declSpecifierSeq is used to deal with conversion of types.
- initDeclaratorList is iterated over to get access to multiple declarations in a single line.
- We use both to allow implicit casting.

Proposed Approach

- **A Sample Parse Tree for Namespace Declarations** :



- namespaceDefinition is the relevant nonterminal.
- Namespaces can be nested and therefore we can have them occuring in subtrees of themselves.

Experimental Results

- **Standard Declarations (supports more types and length modifiers)**:

```cpp
#include <iostream>

int main()
{
    const int a = 5;
    auto b = 4 * 2 + 1;
    char c = 'a', d;
    bool d = true;
    int64_t e = 1;
    unsigned long long int e = d ? 5 : 6;
    printf("%d", e);
}
```

```rust
#![allow(warnings, unused)]
fn main() {
    const a: i32 = 5 as i32;
    let mut b = 4 * 2 + 1;
    let mut c: char = 'a' as char;
    let mut d: char;
    let mut d: bool = true as bool;
    let mut e: i64 = 1 as i64;
    let mut e: u64 = if d { 5 } else { 6 } as u64;
    println!("{}", e);
}
```

- **Enumerations** :

```cpp
#include <iostream>

enum Orientation : int {
  Clockwise = 1,
  Anticlockwise = -1,
  Collinear = 0,
};

int main() {}
```

```rust
#![allow(warnings, unused)]
enum Orientation {
    Clockwise = 1,
    Anticlockwise = -1,
    Collinear = 0,
}
fn main() {}
```

Experimental Results

- **Namespace Declarations :**

```cpp
#include <iostream>

using namespace std;

namespace myapp
{
  namespace xd
  {
    namespace tl {}
    void tr() { cout << "Inside\n"; }
  } // namespace xd
  void pr() { cout << "Hello\n"; }
} // namespace myapp

int main()
{
  myapp::pr();
  using namespace myapp::xd;
  tr();
}
```

```rust
#![allow(warnings, unused)]
use std::*;
mod myapp {
    pub mod xd {
        pub mod tl {}
        pub fn tr() {
            println!("Inside\n");
        }
    }
    pub fn pr() {
        println!("Hello\n");
    }
}
fn main() {
    myapp::pr();
    use myapp::xd::*;
    tr();
}
```

Experimental Results

- **Templates:**

```
template <typename T, typename R> T sum(T x, R y) {
  return x + y;
}
```

```
fn sum<T, R>(mut x: T, mut y: R) -> T {
    return x + y;
}
```

- **Class declarations with separate method definitions:**

```
#include <iostream>
template <typename T> class TemplExample {
private:
  T obj;
  int size;

public:
  TemplExample(T o, int s);
  void print();
};

template <typename T> TemplExample<T>::TemplExample(T o, int s) {
  obj = o;
  size = s;
  for (int i = 0; i < size; i++)
    obj += o;
}

template <typename T> void TemplExample<T>::print() {
  std::cout << " " << obj;
}
```

```
#![allow(warnings, unused)]
#[derive(Default)]
pub struct TemplExample<T> {
    obj: T,
    size: i32,
}
impl<T> TemplExample<T> {
    fn new(mut o: T, mut s: i32) -> TemplExample<T> {
        obj = o as i32;
        size = s;
        let mut i: i32 = 0 as i32;
        while i < size {
            obj += o;
            i += 1;
        }
        /*
            This is a constructor method.
            Please appropriate members to the struct constructor as per your logic.
            Currently the constructor returns a struct with all the defaults for the
        */
        TemplExample {
            ..Default::default()
        }
    }
}
impl<T> TemplExample<T> {
    fn print(&mut self) {
        std::println!(" {}", self.obj);
    }
}
```

Experimental Results

- **Multi-dimensional Arrays:**

```cpp
#include <iostream>

int main() {
    int D[20];
    int twoD[20][30];
    int threeD[20][30][40];
}
```

```rust
#![allow(warnings, unused)]
fn main() {
    let mut D: [i32; 20];
    let mut twoD: [[i32; 30]; 20];
    let mut threeD: [[[i32; 40]; 30]; 20];
}
```

- **Lambdas:**

```cpp
int main() {
  int y = 2;
  auto mul_by_val = [=](int x) { return x * y; };
  auto mul_by_ref = [&, y](int x) -> int { return x * y; };
}
```

```rust
#![allow(warnings, unused)]
fn main() {
    let mut y: i32 = 2 as i32;
    let mut mul_by_val = move |mut x: i32| { return x * y; };
    let mut mul_by_ref = |mut x: i32| -> i32 { return x * y; };
}
```

Existing Transpilations (Summary):

Construct	Not Supported	Supported
Main function(CLI arguments not supported)		✔
Simple Declarations (fixed size variables and arrays)		✔
Implicit and Explicit casting		✔
Functions		✔
Pointers	✔	
Structs and classes (partial i.e. lack of abstraction and inheritance)		✔
Exception handling	✔	
NULL values	✔	
Namespaces (Functions but not variables) (Nested Namespaces as well)		✔
Library Functions (Header files)	✔	

Observations and Challenges

- **Major Observations / Conclusions & Challenges :**

Challenges Faced:

- Using same visitor function for multiple purposes . For example, modifying visitors corresponding to common non-terminal have long-reaching consequences.
- Learning rust syntax and semantics.
- Transpiling constructs that involve higher level constructs like inheritance.
- No easy one to one mapping available for memory management and garbage collection in rust and C++.

Observations:

- Converting C or C++ library functions will be necessary
- Data types which do not have the Copy trait can cause borrow checker issues.
- Direct translation of code does not take advantage of some of Rust features, eg: match statements,
- C code involving pointers may not have an equivalent code in safe Rust.

Further Plan to Complete Project

- **Final Probable Deliverables** :
  (Discuss in the form of bullets, what are the next steps to complete the solution, any road blocks / bottlenecks, any support needed from SRIB)

- Direct conversion of C/C++ pointers to Rust safe pointers
- Addition of Trait declaration to the generated Rust code
- Conversion of C/C++ standard library functions to equivalent Rust whenever possible
- Enhanced error messages for failed conversions

- **IP Target / Plan** :
  (Any possibility of papers / patentable ideas / innovative aspects that can lead to patentable ideas)

- A paper discussing our approach of transpilation and the extent of transpilation achieved.
- A paper looking into leveraging rusts borrow checker for avoidance of race conditions and detection of unhandled conditions in existing C/C++ code
- Publishing the transpiler developed

Further Plan to Complete Project

- **Completion Plan**:
  (High level plan to complete the project in next 8 weeks after review, in format below)

Week 1 to 2	• Step 1: Exploration of C direct pointer conversion methods • Step 2: Reduction of conflicts with borrow checker for c pointer conversion
Week 3 to 4	• Step 1: Testing C++ code conversion and fixing code generation • Step 2: Detection and declaration of required Rust Traits
Week 5 to 6	• Step 1: Implementing basic work around for inheritance • Step 2: Conversion of C/C++ library functions
Week 7 to 8	• Step 1: Addition of error messages and testing the conversions • Step 2: Removal of bugs and finalizing the project

- **Challenges Anticipated:**

  - Rust's lack of inheritance can hamper C++ code conversion evaluation
  - Passing pointers as function parameters can cause borrow checking issues

- **Git Upload details:** Current version of the transpiler has been uploaded. Git Repo

Thank you