Re-Implementing P0 with Rust

McMaster University

Group 7: Kumail Naqvi, Ghazi Salam, Nico Stepan

Overview

This project's goal was to re-implement the Python version of the P0 compiler in Rust. Compilers need to be fast and efficient as they control the speed of the language at the most base level so implementing P0 in a systems language instead of Python was an attempt to speed up the language and compare the efficiency of the compiler on different languages.

Why Rust?

Rust is a relatively new systems level multi-paradigm language. Its performance is comparable to C/C++.

- ✅ Generic
- ✅ Imperative
- ✅ Structured
- ✅ Functional

Rust's main features are its memory safety and predictable run time behaviour. All memory allocation is done by the user which leads to safety against null pointers and race conditions. Rust uses 'ownership' which allows one to write code without a garbage collector or segmentation faults. It also has an advanced compiler that statically checks for errors.

Safey Guarantees

- Segment Fault ❌
- Runtime Crash ❌
- Dangling Pointers ❌
- Out-of-bound accesses ❌
- Data Race ❌

Our Experience

Code statistics: ~2,400 LOC
Test P0 Programs: 11
Run on: macOS 10.15.3 with a 2.6 GHz 6-Core Intel Core i7 Processor and 16GB of 2400MHz DDR4 onboard memory

Design Choices

Unlike Python, Rust is not a dynamically typed language and so our P0 implementation does not create individual classes and constructs for each type. Instead, our design structure was chosen as follows:

```rust
pub struct Symbol {
    pub name: String,
    pub lev: i32,
    pub tp: PrimitiveTypes,
    pub val: i32,
    pub par: Vec<String>,
    pub base: Base,
    pub lower: i32,
    pub length: i32,
    pub adr: i32,
    pub size: i32,
    pub fields: Vec<Field>,
    pub type_name: String,
}
```

```rust
x = ST::Symbol {
    type_name: "Const".to_string(),
    val: SC::val_number(),
    tp: ST::PrimitiveTypes {
        type_name: "Int".to_string(),
        ..Default::default()
    },
    ..Default::default()
};
x = CGwat::genConst(x);
```

```rust
pub fn genAssign(x: ST::Symbol, y: ST::Symbol) {
    if x.type_name == "Var".to_string() {
        if x.lev == -2 {
            asm_push((r"i32.const ".to_owned()
                + &x.adr.to_string()).to_string());
        }
        loadItem(&y);
```

The scanner is implemented such that a global symbol value is used to distinguish between a string and a number:

```rust
static ref val: Mutex<ValType> =
        Mutex::new(ValType {
        number: 0 as i32,
        string: "".to_string(),
        none: 1 as i32,
        type_name: "none".to_string(),
    }
);
```

```rust
val.lock().unwrap().string = source
    .lock()
    .unwrap()
    .chars()
    .skip(start as usize)
    .take((index - start - 1) as usize)
    .collect();
```

CGwat makes use of a global array to store each of the WebAssembly lines which at the end of the program is printed in the console. The following are examples of the genBinaryOp and genProgExit functions:

```rust
pub fn genBinaryOp(
    op: i32,
    mut x: ST::Symbol,
    : ST::Symbol)
    -> ST::Symbol {
    if op == PLUS || op == MINUS
    || op == TIMES || op == DIV || op == MOD {
        loadItem(&x);
        loadItem(&y);
        match op {
            TIMES => asm_push(r"i32.mul".to_st
            MINUS => asm_push(r"i32.sub".to_st
            PLUS => asm_push(r"i32.add".to_str
```

```rust
pub fn genProgExit() -> String {
    let mut _mem = unsafe { memsize };
    unsafe {
        asm_push(
            (")\n(memory ".to_owned()
                + &(memsize / i32::pow(2, 16) + 1)
                    .to_string()
                + ")\n(start $program)\n)")
                    .to_string(),
        );
    }
    let asm = ASM.lock().unwrap();
    let masterString = asm.join("\n");
```

Testing and Results

Correctness was asserted by manually checking that the compiled P0 code from Rust was equivalent to that from the Python implementation. Runtime in Rust was measured via the *std::time::Instant*. The below charts summarize our runtime results:

P0 Program	Rust (ms)	Python (ms)	Go (Group 11) (ms)
A (5 LOC)	0.242	1.043	0.967
B (10 LOC)	0.404	1.422	1.447
D (10 LOC)	0.281	1.180	1.738
C (15 LOC)	0.592	1.543	1.299
E (50 LOC)	3.157	6.927	1.599

P0 Program	Rust (s)	Python (s)	Go (Group 4) (s)
500 LOC	0.048	0.041	0.005
1000 LOC	0.177	0.063	0.011
2000 LOC	0.653	0.124	0.029
3000 LOC	1.593	0.186	0.061
4000 LOC	2.766	0.247	0.113
5000 LOC	4.070	0.402	0.185

Why is Rust so much slower? The P0 compiler relies on global variables that fight against memory safety so Rust does not allow them. Our implementation forced us to use mutex locks and unsafe code blocks which lead to a slowdown of the compiler's parsing ability.

In conclusion, our Rust P0 implementation is better than Python's for P0 programs under 500 LOC. Further research and experience with the language is required to utilize the full potential of Rust.

An Outlook

Missing in our current implementation is a full implementation of procedures (only write(), read(), and writeln() work), and arrays and records testing.