C4 Rust Implementation: Comparative Analysis

Introduction

The C4 compiler, created by Robert Swierczek, is a minimalist C compiler implemented in just four functions. This report examines a Rust reimplementation of C4, analyzing how Rust's safety features influenced the design while maintaining compatibility with the original.

Memory Safety Enhancements

Ownership and Borrowing

The original C4 implementation relies on raw pointers and manual memory management:

```
if (!(sym = malloc(poolsz))) { printf("could not malloc(%d) symbol area\n", poolsz); return -1; } if (!(data = malloc(poolsz))) { printf("could not malloc(%d) data area\n", poolsz); return -1; }
```

The Rust implementation leverages Rust's ownership model to eliminate these risks:

```
pub struct C4 {
    pub symbols: Vec<Symbol>, // Symbol table
pub text: Vec<i32>,       // Text segment     pub
data: Vec<i32>,       // Data segment
    // ...
}
```

By using `Vec<T>` instead of raw pointers, the Rust implementation guarantees memory is automatically freed, prevents buffer overflows through bounds checking, and eliminates dangling pointers through lifetime management.

Type Safety with Enums

The C implementation uses magic numbers for tokens and instructions, while the Rust version improves type safety with proper enums:

```
[derive(Debug, PartialEq, Clone, Copy)] pub enum TokenType {
    Num = 128, Float = 257, Fun, Sys, Glo, Loc, Id, // ... }
```

This approach prevents type confusion and makes the code more self-documenting.

Design Adaptations

Structured Error Handling

The C4 C implementation handles errors by printing messages and calling `exit(-1)`:

```
if (tk != Id) { printf("%d: bad global declaration\n", line); exit(-1); }
```

The Rust version uses more structured error handling with Result types:

```
pub fn main() -> io::Result<()> {    let mut file = File::open(&args[1])?;
    // ...
}
```

Symbol Table Implementation

The C implementation uses a flat array for the symbol table with fixed offsets, while the Rust version uses a proper struct:

```
#[derive(Debug, Clone)] pub struct Symbol {     pub token: TokenType,     pub hash: i32,     pub name: String,
    // ...
}
```

This improves code readability and prevents offset errors.

Implementation Challenges

C-Compatible Memory Layout

Maintaining compatibility with C's memory model was challenging. The Rust implementation had to carefully manage memory layouts:

```
// In C: if (id[Class] == Loc) { *++e = LEA; *++e = loc - id[Val]; } //
In Rust:
if self.symbols[symbol_idx as usize].class == TokenType::Loc as i32 {
self.text.push(Instruction::LEA as i32);
    self.text.push(self.index_of_bp - self.symbols[symbol_idx as usize].value); }
```

Replicating C's Implicit Behaviors

The C implementation relies on implicit type conversions and pointer arithmetic that don't directly translate to Rust:

```
// C code: *++e = IMM; *++e = ival; next(); ty = INT;

// Rust equivalent: self.text.push(Instruction::IMM
as i32); self.text.push(self.token_val);
self.next(); self.expr_type
= INT;
```

Virtual Machine Implementation

Implementing the virtual machine in Rust required balancing memory safety with performance:

```
// C: else if (i == LI) a = *(int *)a;

// Rust:
op if op == Instruction::LI as i32 => {
self.ax = self.stack[self.ax as usize];
},
```

Performance Considerations

Multiple components determine how fast the Rust version operates:

1. The boundary protection mechanism in Rust stops buffer overflows during runtime execution though it requires a minor cost. Recursive array operations occur often in both lexer and parser sections.

2. The RAII structure in Rust removes human-caused memory management errors yet leads to different methods of memory allocation. The Rust implementation adopts `Vec<T>` vectors that exhibit memory reallocation behavior when their capacity expands although C arrays maintain fixed capacity dimensions.

3. Program execution efficiency improves through the implementation of enums for instructions because they enable better branch prediction and potential code generation efficiency. The Rust implementation's performance matches approximately the C version under typical conditions and might offer better outcomes because of Rust's optimization potential.

Conclusion

Rust implementation of C4 achieves full compatibility with the C original code yet provides enhanced memory safety as well as improved code organization standards. The main difficulties emerged as a result of converting C's unregulated memory access control to Rust's protective memory management system.

Key improvements in the Rust version include:

- Better type safety through proper enums and structs
- More structured error handling
- Improved code organization and readability

The study shows how Rust enables safety mechanisms for systems programming to benefit from C-like control while ensuring program reliability at comparable performance levels.