C4 Rust Implementation: Comparative Analysis

Introduction

This report compares our Rust implementation of the C4 compiler with the original C version created by Robert Swierczek. The C4 compiler is a minimalist, self-hosting C compiler that implements a subset of the C language in fewer than 1000 lines of code. Our goal was to rewrite this compiler in Rust while preserving its functionality and self-hosting capability, leveraging Rust's safety features and modern programming paradigms.

Memory Safety and Rust Features

Memory Management

The original C4 implementation relies heavily on manual memory management with direct calls to malloc() and free(), along with raw pointer manipulation. This approach is error-prone and can lead to memory leaks, buffer overflows, and use-after-free bugs.

Our Rust implementation leverages Rust's ownership system to eliminate these issues:

rust
// Original C4 approach (simplified):
if (!(sym = malloc(poolsz))) { printf("could not malloc(%d) symbol area\n", poolsz); return -1; }
if (!(le = e = malloc(poolsz))) { printf("could not malloc(%d) text area\n", poolsz); return -1; }
if (!(data = malloc(poolsz))) { printf("could not malloc(%d) data area\n", poolsz); return -1; }
// ...later...
free(sym); free(e); free(data);

// Rust approach:
let symbol_table = SymbolTable::new();
let mut code = Vec::new();
let mut data_segment = Vec::new();
// Memory is automatically freed when variables go out of scope
By using Rust's Vec, HashMap, and other standard library collections, we eliminate the need for explicit memory allocation and deallocation. Memory is automatically reclaimed when variables go out of scope, preventing leaks and use-after-free bugs.

Bounds Checking

A significant safety improvement in our implementation is automatic bounds checking:

rust
```
// Original C4 (no bounds checking):
void next() {
  // ...
  while (tk = *p) {
    ++p; // Potential buffer overrun
    // ...
  }
}

// Rust implementation (with bounds checking):
fn next_token(&mut self) -> Result<Token, CompilerError> {
    // ...
    while let Some(ch) = self.current_char() {
        // Safe access that returns None at end of string
        // ...
    }
    // ...
}
```
This prevents buffer overflows, a common source of security vulnerabilities in C programs.

Null Safety

The C4 compiler frequently uses null pointers and assumes that certain operations won't produce null results. Our Rust implementation replaces these with Option types, ensuring that null cases are explicitly handled:

rust
```
// Original C4:
if (id[Class] == Loc) { printf("%d: duplicate local definition\n", line); exit(-1); }
id[HClass] = id[Class]; id[Class] = Loc;

// Rust approach:
if self.symbol_table.exists_in_current_scope(&var_name) {
    return Err(CompilerError::ParserError(
        format!("Duplicate local definition: {} at line {}", var_name, self.lexer.line())
    ));
}
```

Error Handling

The original C4 handles errors by printing messages and immediately exiting the program:

c
```
if (tk != Id) { printf("%d: bad global declaration\n", line); return -1; }
if (id[Class]) { printf("%d: duplicate global definition\n", line); return -1; }
```
Our Rust implementation uses Result types for proper error propagation:

rust
```
fn parse_global_variable(&mut self) -> Result<(), CompilerError> {
    // ...
    if self.current_token.token_type != TokenType::Id {
        return Err(CompilerError::ParserError(
            format!("Expected identifier, got {:?}", self.current_token.token_type)
        ));
    }
    // ...
}
```
This approach provides several benefits:

- Errors can be handled at the appropriate level rather than immediately exiting
- More detailed error information can be provided
- Different error types can be distinguished and handled differently
- The compiler becomes more usable as a library

Type Safety

We replaced C4's integer-based token and type system with proper Rust enums:

c
```
// C4 approach:
enum {
  Num = 128, Fun, Sys, Glo, Loc, Id,
  Char, Else, Enum, If, Int, Return, Sizeof, While,
  // ...
};

// Rust approach:
pub enum TokenType {
    // EOF sentinel
    Eof,
```

```
    // Keywords
    Char, Else, Enum, If, Int, Return, Sizeof, While,

    // Variable/function classes
    Num, Str, Fun, Sys, Glo, Loc, Id,

    // Operators (in precedence order)
    Assign, Cond, Lor, Lan, Or, Xor, And, Eq, Ne, Lt, Gt, Le, Ge, Shl, Shr, Add, Sub, Mul, Div,
Mod, Inc, Dec, Brak,
    // ...
}
```

This ensures that the compiler can catch type errors at compile time rather than runtime, reducing bugs and improving code clarity.

Implementation Challenges

Global State

The original C4 uses global variables extensively, which makes the code less modular and harder to test. Converting this to Rust's ownership model presented challenges:

```
c// C4 global state:
char *p, *lp, *data;
int *e, *le, *id, *sym, tk, ival, ty, loc, line, src, debug;

// Rust encapsulated state:
pub struct Parser {
    lexer: Lexer,
    code: Vec<i64>,
    current_token: Token,
    symbol_table: SymbolTable,
    data_segment: Vec<u8>,
    // ...
}
```

We had to carefully analyze how these global variables were used and ensure that our encapsulated state maintained the same behavior while being safer and more maintainable.

Replacing Global Mutable Pointers

A particular challenge was replacing the global pointer p in C4's next() function, which is used to track the current position in the source code:

```
c// In the original C4:
char *p, *lp; // global pointers to track source position

void next() {
  while (tk = *p) {
    ++p;
```

```
    // Process tokens...
  }
}
```

This approach relies on global mutable state, which Rust discourages due to its ownership and borrowing rules. In our Rust implementation, we encapsulated this state within the Lexer struct and provided methods to safely access and manipulate it:

```
rust// In our Rust implementation:
pub struct Lexer {
    source: String,
    position: usize,    // Current position in source
    line_position: usize, // Line start position
    line: usize,        // Current line number

    current: Token,     // Current token information
    // ...
}

impl Lexer {
    // Get current character at position
    fn current_char(&self) -> Option<char> {
        self.source.chars().nth(self.position)
    }

    // Advance position
    fn advance(&mut self) -> Option<char> {
        let current = self.current_char();
        self.position += 1;

        // Track line numbers for error reporting
        if current == Some('\n') {
            self.line += 1;
            self.line_position = self.position;
        }

        current
    }

    // Get next token (similar to C4's next() function)
    pub fn next_token(&mut self) -> Result<Token, CompilerError> {
        // Skip whitespace
        self.skip_whitespace();

        // Get current character
        match self.current_char() {
```

```
            None => {
                // End of file
                Ok(Token {
                    token_type: TokenType::Eof,
                    value: None,
                    name: None,
                })
            },
            Some(c) => {
                // Process character and create appropriate token
                self.advance(); // Advance position (equivalent to ++p in C4)

                // Token processing logic...
                // ...
            }
        }
    }
}
```

This approach offers several advantages:

- Encapsulation: Source position state is contained within the Lexer
- Memory Safety: No raw pointers or potential buffer overflows
- Error Handling: Returns Result instead of modifying globals
- Testability: Can create multiple independent lexers for testing
- Thread Safety: No shared mutable state between threads

By encapsulating the state that was previously global, we maintain the same functionality while adhering to Rust's ownership and borrowing rules, resulting in safer and more maintainable code.

Symbol Table Management

C4's symbol table uses a flat array with special handling for scope and shadowing:

c// C4 approach to restore symbols after function scope exit:

```
id = sym; // unwind symbol table locals
while (id[Tk]) {
  if (id[Class] == Loc) {
    id[Class] = id[HClass];
    id[Type] = id[HType];
    id[Val] = id[HVal];
  }
  id = id + Idsz;
}
```

Our approach uses a more structured representation with explicit scope management:

rust// Rust approach with explicit scope handling:

```
pub fn exit_scope(&mut self) {
```

```
    if self.scopes.len() <= 1 {
        // Don't exit global scope
        return;
    }

    // Get the start index of the current scope
    let start_index = self.scopes.pop().unwrap();

    // Remove scope entries from name map
    let scope_level = self.scopes.len();

    // Remove entries from current scope
    let keys_to_remove: Vec<String> = self.name_map.keys()
        .filter(|k| k.starts_with(&format!("{}:", scope_level)))
        .cloned()
        .collect();

    for key in keys_to_remove {
        self.name_map.remove(&key);
    }

    // Keep only symbols from outer scopes
    self.symbols.truncate(start_index);
}
```

Instruction Encoding

C4 uses integers directly for instruction encoding, which is efficient but lacks type safety. We introduced proper enums while maintaining the same binary representation:

```
rustpub enum Opcode {
    LEA, IMM, JMP, JSR, BZ, BNZ, ENT, ADJ, LEV, LI, LC, SI, SC, PSH,
    OR, XOR, AND, EQ, NE, LT, GT, LE, GE, SHL, SHR, ADD, SUB, MUL, DIV, MOD,
    OPEN, READ, CLOS, PRTF, MALC, FREE, MSET, MCMP, EXIT,
}

// During code emission:
fn emit(&mut self, code: i64) -> usize {
    let pos = self.code.len();
    self.code.push(code);
    pos
}

// Usage:
self.emit(Opcode::ADD as i64);
```

This preserves compatibility while improving code readability and safety.

Conclusion

Our Rust implementation of the C4 compiler achieves several important goals:

1. Maintains Functional Compatibility: It compiles the same subset of C as the original, including self-hosting capability.
2. Improves Safety: It leverages Rust's memory safety, type system, and error handling to eliminate entire classes of bugs.
3. Enhances Performance: Despite the larger binary size, it offers better runtime performance and memory efficiency.
4. Improves Maintainability: The modular structure, clear API boundaries, and comprehensive error handling make the code easier to understand and extend.

These improvements demonstrate how Rust's safety features can be applied to systems programming tasks that traditionally relied on C, without sacrificing performance or compatibility. The challenges we encountered during implementation were primarily related to reconciling C4's global state and direct memory manipulation with Rust's ownership model and safety guarantees.
This project serves as a case study in migrating legacy C codebases to Rust, highlighting both the benefits of such migrations and the strategies for maintaining compatibility while improving safety and maintainability.