Thesis Dissertation

TRANSLATING C TO RUST MIR

Georgia Michail

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2025

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

Translating C to Rust MIR

Georgia Michail

Supervisor

Dr. Elias Athanasopoulos

Thesis submitted in partial fulfillment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2025

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Elias Athanasopoulos, for his invaluable guidance, support, and the wealth of knowledge he has shared with me throughout this past year. His mentorship has played a crucial role in the completion of this thesis.

I am also very thankful to my family and close friends for all the support and understanding through the course of my studies.

Finally, I am very grateful for all the experience and knowledge I gained over the past four years. I am closing this chapter of my life wiser, happier, and with gratitude.

ABSTRACT

This thesis describes a static analysis and translation environment that converts C code into a Rust-style Mid-level Intermediate Representation (MIR). C remains a major language for systems programming but lacks essential safety guarantees such as ownership, lifetime enforcement, and overflow detection. Rust, by contrast, enforces these guarantees through MIR, a structured compiler-level representation used for borrow checking and early optimizations. Our work describes a prototype tool that explores traversal of C sources by the use of Clang's AST and LibTooling and generates MIR-style output. The tool's generated output maintains control flow, scope of variables, temporary registers, and semantics of formatted printing, closely recreating the structure and behavior of MIR outputted by the Rust compiler. The tool implements key C constructs such as functions, expressions, conditionals, and loops, and has mechanisms for safe arithmetic and printf translation. The evaluation results show that the tool produces correct MIR-style output for a wide set of C programs, making it a solid starting point for future work in static analysis and secure code translation.

Contents

Chapter 1

Introduction

Safety, performance, and reliability are more critical than ever in the modern era of systems programming. Languages like C have served as the backbone of low-level software for decades, yet they bring inherent risks: memory errors, data races, and undefined behavior. Rust has emerged as a powerful alternative, offering memory safety guarantees without sacrificing speed by managing memory in an unconventional way. Rather than relying on garbage collection or explicit memory allocation, Rust enforces an ownership model through a compiler-defined set of rules. As a result, memory and thread safety are statically enforced at compile time, thereby eliminating entire classes of bugs that C leaves to be addressed manually by the programmer. Yet the challenge remains: how do we reconcile the massive base of existing C code with the safety standards of Rust? This thesis addresses that question by proposing a novel translation framework that brings the structural rigor of Rust's MIR (Mid-level Intermediate Representation) to legacy C code.

The main research problem addressed in this thesis is how to semantically convert C code into a representation that mimics Rust's Mid-level Representation (MIR), so preserving its control flow, variable lifetimes, formatting, and overflow checks. Unlike source-to-source translators that convert C code directly into Rust syntax, our method focuses on generating MIR-like output to enable static analysis, optimization modeling, and possible integration into compiler backends.

Significant difficulties arise when we are trying to translate C's semantics which lack inherent concepts of ownership, borrow checking, and lifetime enforcement, into Rust's strict safety model. Many times, current tools, such as C2Rust, generate Rust code that relies mostly on unsafe blocks to manage memory operations and raw pointers, so as a result they limit the safety guarantees that Rust aims to provide [11]. According to Emre et al. (2021), aliasing in C code, multiple pointers referring to the same memory, creates important challenges to automated ownership inference when translating to Rust. In many cases, developers need to step in manually to ensure safety [19]. Hardekopf et al.

(2021) emphasizes on the difficulty involved in achieving memory safety after translation and offers transformation patterns to turn the unsafe Rust code into safer alternatives [20]. Finally, Silva et al. investigates the use of borrow-checking concepts on C code, showing how static analysis might enforce Rust-like safety guarantees in C programs [21].

However, despite the related works we already mentioned, there is still a need for tools that can directly convert C code into MIR-like code, enabling easier analysis and potentially work with Rust's compiler infrastructure. By focusing on generating MIR-style output, this thesis aims to fulfill the need for such tools and provide a pathway for legacy C code to benefit from Rust's safety mechanisms without requiring major manual refactoring or unsafe solutions.

Converting legacy C code into MIR-style representation has important academic and practical advantages. C remains widely used in embedded systems, OS kernels, and critical infrastructure, areas where safety lapses can be catastrophic. A 2019 Microsoft report found that the majority of vulnerabilities Microsoft assigns a CVE each year continue to be memory safety issues, primarily stemming from C and C++ codebases [24]. Similarly, a study by Google found that 70% of all "severe security bugs" in Chromium were caused by memory safety problems [25].

Moreover, Rust's compiler heavily relies on MIR for borrow checking and early optimizations and exposing these benefits to C code via translation enables research on cross-language safety enforcement. In doing so, this tool may serve as a research platform for future work in hybrid analysis, safe interoperability, and even automated rewriting of unsafe legacy systems.

Translating from C to MIR is not easy. The concepts of ownership, borrow checking, or lifetime checking, features essential to Rust's safety guarantees are not native to C code. It takes careful static inference, approximations, and assumptions to map raw C pointers and memory to a structured intermediate format. The main challenge we recognized during the creation of this tool, is to keep the control flow and semantics of the original C code while still generating valid and analyzable MIR-style output.

While previous tools like C2Rust focused on directly translating C to Rust, often require extra information from the developer or make educated guesses about the code's

behavior that may lead to unsafe code. Our work is unique in targeting MIR directly, thus avoiding the complexities of generating Rust source instead enabling reasoning at the compiler IR level, where true safety checks occur.

This thesis makes the following contributions:

- A prototype C++ tool under development, built using Clang's AST and LibTooling, designed to traverse C code and emit a Rust MIR-style intermediate representation.
- Support for major language constructs, including functions, variable declarations, binary operations, while loops, partial conditionals, and formatted output.
- Automatic promotion of static segments, construction of formatting arrays, and generation of `Arguments::new_v1` and `new_v1_formatted` calls, mimicking Rust's `println!` machinery.
- Safety checks for arithmetic, such as overflow assertions using `AddWithOverflow`, `DivWithOverflow`, `MulWithOverflow`, `SubWithOverflow` and runtime division-by-zero guards.
- A structured basic block system, preserving the control flow, return paths, and deferred assignments, just like actual MIR.
- A scope-aware variable tracking system, with accurate from C names to MIR registers and inclusion of debug metadata.
- A thorough evaluation suite, comparing the output of thel tool with real Rust MIR using equivalent Rust programs compiled with `-emit=mir`.

Chapter 2

Background

This chapter provides the necessary background on the Mid-level Intermediate Representation (MIR) used by the Rust compiler, and on the Clang AST framework used to traverse and analyze C programs. While both C and Rust are well-known systems languages, this work focuses on bridging their representations through static analysis and transformation.

2.1 Mid-level Intermediate Representation (MIR)

2.1.1 Introduction to MIR

Mid-level Intermediate Representation (MIR) represents Rust's simplified, control-flow-oriented form of code, which is constructed from the High-level Intermediate Representation (HIR), the primary intermediate form that the Rust compiler, rustc, uses for the majority of its internal processing and language evolution. MIR serves as an intermediate stage between the HIR and LLVM IR (see figure 1).

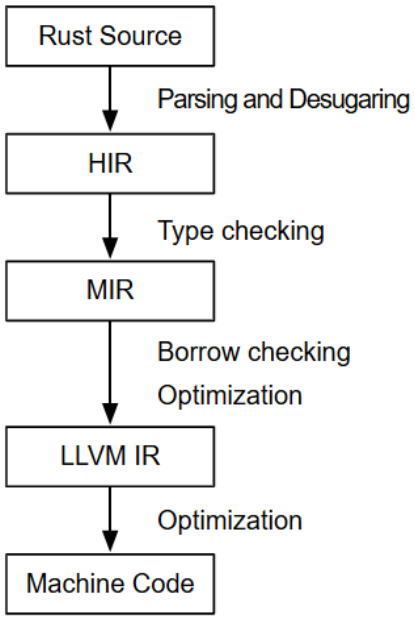


Figure 1: Rust compilation pipeline

For developers, MIR brings several practical benefits including more flexible borrowing, improves type checking, and helps the compiler catch issues earlier. It also achieves faster compilation by allowing incremental builds and early Rust-specific optimizations, even before going to LLVM. Execution speed also benefits from MIR through smarter handling of drop. Internally, MIR simplifies compiler's logic by handling key transformations like match evaluation and drop semantics, reducing redundancy across stages. Since it has a uniform, low-level structure makes it easier to develop and maintain complex features and set the stage for developing advanced compilers in the future.

MIR consists of method signatures that have parameter and variable declarations, scopes, and basic blocks that define control flow through sequential statements and a single terminating instruction. The basic blocks include assertions for safety, assignments for value handling, and metadata like method signatures, scopes, and parameters. Its structured and transparent format makes MIR ideal for analysis, optimization, and formal reasoning in systems programming.

```
fn add(_1: i32, _2: i32) -> i32 {
    debug a => _1;
    debug b => _2;
    let mut _0: i32;
    let mut _3: (i32, bool);

    bb0: {
        _3 = AddWithOverflow(copy _1, copy _2);
        assert(!move (_3.1: bool), "attempt to compute `{} + {}`, which
would overflow", copy _1, copy _2) -> [success: bb1, unwind continue];
    }

    bb1: {
        _0 = move (_3.0: i32);
        return;
    }
}
```

2.1.2 Why Use MIR for Analysis and Translation

One of the main reasons this project adopts Rust's MIR as a model for translating and analyzing C code is that MIR is the level where the Rust borrow checker operates. The borrow checker is responsible for enforcing ownership, borrowing, and lifetime rules in Rust programs. It ensures that memory accesses are valid, non-overlapping, and free of data races, but critically, this validation does not occur at the high-level source or low-level LLVM IR stages. Instead, the borrow checker analyses the MIR form of the code, which is structured in a way that makes such analysis both tractable and precise.

MIR is more beneficial in many was than lower-level code like LLVM IR and three-address format. Unlike LLVM IR, which is mostly focused on low-level details of hardware unlike MIR that is concerned with features found in Rust such as moves, scopes, drop logic, and references. These features are essential for understanding how memory is used correctly and for safely analyzing or converting programs. Additionally, MIR's control-flow structure makes the program's logic easier to understand and modify compared to the more flattened and low-level style of three-address code.

By converting C code into a MIR-like representation, this thesis makes it possible to analyse at the LLVM level, such as ownership tracking, early borrow checking, and variable lifetime inference. Although C does not have an ownership model, representing C programs in a MIR-like format allows the application of Rust-style safety principles, including the possibility of identifying violations similar to those caught by the borrow checker. This makes MIR not only a suitable but an ideal target format for static analysis, transformation, and future safety checking of legacy C code.

2.2 Abstract Syntax Trees and Clang Tooling

2.2.1 The Role of ASTs in Compilers

An Abstract Syntax Tree (AST) is a core data structure used in compilers to present source code in an ordered tree-like form. The AST's logic is straightforward since every expression, statement, or declaration map to an AST node. ASTs have a hierarchical

structure, as shown in the figure below, they do not include syntactic details and preserve the original program's logical order. More specifically, nodes represent code constructs, whereas edges represent relationships between the constructs.

ASTs are useful for analyzing code statically without executing it, because they let us traverse the code and explore the structure of the code directly. This makes them great for catching bug early. Additionally, because ASTs allow changes in the code's structure without compromising functionality, we can efficiently transform and refactor code using them.

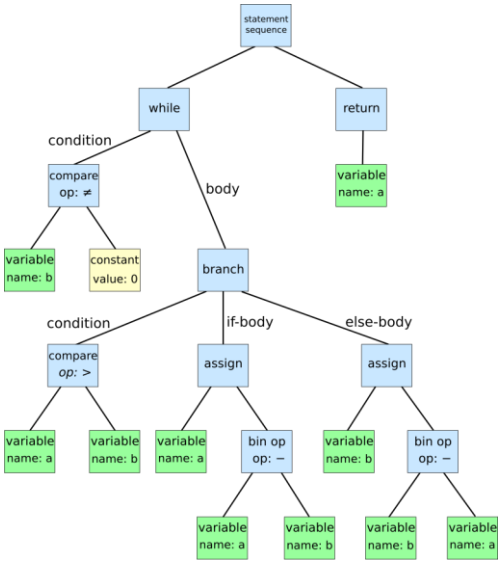


Figure 3: AST example

2.2.2 Clang's AST and LibTooling Framework

Clang's AST constitutes a detailed and structured representation of source code written in C, C++, or Objective-C. The most basic node types include `Decl` and `Stmt`. Using Clang's `RecursiveASTVisitor` interface, developers can perform custom analysis by overriding methods such as `VisitFunctionDecl` or `VisitVarDecl` to walk through the AST. Nevertheless, this framework equips tools with strong capabilities for performing static analysis, modification, and even restructuring, by representing the program's semantics in a well-organized and accessible way.

To aid the creation of such tools Clang implemented LibTooling. LibTooling provides access to the compiler's inner workings, allowing developers to parse code, run diagnostics, and generate abstract syntax trees (ASTs). Particularly, this project utilizes LibTooling to retrieve the structures of C code and translate them into MIR-like

intermediate form, allowing analysis and modifications with precise control flow. The adaptability of the Clang Abstract Syntax Tree, along with the use of LibTooling makes them ideal for creating applications that convert source code into an intermediary format (IR), similar to what our project performs.

Chapter 3

System Architecture

3.1 System Overview

The aim of this thesis is to translate C code to the Rust-style Mid-level Intermediate Representation (MIR) output, driven by the broader goal of enabling structured and semantically aware translation from C to Rust. The tool we are implementing leverages Clang's LibTooling and AST Libraries and is written in C++ since it is directly built on top of Clang's LibTooling and AST infrastructure, which is itself implemented in C++. Our approach does not attempt to generate executable Rust code directly, instead it focuses on translating C to MIR by reproducing the structure and semantics of MIR, while iterating and analyzing the C Abstract Syntax Tree (AST). The syntactic and semantic details required from our tool to generate the MIR-style output that mimics the Rust compiler are produced by traversing the Abstract Syntax Tree (AST) of the C source files. The tool parses C source files, traverses their AST, and emits MIR-like output that captures the control flow, variable declarations, operations, and debug data. Every line of C code is analyzed and handled in the same way that the equivalent Rust code would be processed. For example, a line like `int c;` in C is treated as if it were `let mut c: i32;` in Rust, closely mimicking the behavior of the Rust compiler. Additionally, since the Rust compiler performs a range of optimizations, these must also be considered during the mimicking stage. The final output aims to resemble what the Rust compiler would produce internally following type-checking, but prior to lowering to LLVM IR.

3.2 Architecture Components

In the following subchapter, we are going to analyse in depth the architecture of each major component within the translation tool.

3.2.1 Frontend Driver and Tool Initialization

The Frontend Driver and Tool Initialization component is responsible for setting up the analysis pipeline. The system starts from the main function and initializes Clang's command-line parser and tooling pipeline by using `CommonOptionsParser` for processing the arguments, configuring ClangTool, and launching the `CustomFrontendAction` to initiate the AST analysis.

```
auto ExpectedParser =CommonOptionsParser::create(argc,argv,ToolCategory);
  CommonOptionsParser &OptionsParser = ExpectedParser.get();
  ClangTool Tool(OptionsParser.getCompilations(),
                 OptionsParser.getSourcePathList());
  return Tool.run(newFrontendActionFactory<CustomFrontendAction>()
                      .get()); // call custom action on frontend which is
to run the consumer
```

3.2.2 Frontend Action and AST Consumer

Once the ClangTool is launched, class `CustomFrontendAction` is invoked and overrides the `CreateASTConsumer` method. This is where our translation pipeline connects to Clang's internal AST infrastructure. The AST consumer processes the parsed translation unit and internally creates an instance of the `MyASTVisitor` class, the core of our translation logic.

3.2.3 AST Traversal and Translation Logic

The core component of the tool, `MyASTVisitor`, utilizes Clang's `RecursiveASTVisitor` to traverse the AST generated from the C source code node by node. In this component, most of the system's logic resides, including variable and

scope management, register allocation, and basic blocks control. Struct `variableInfo`, which has information about every C variable, including its initial value, equivalent MIR name, type, and scope depth, is one of our several helper structures set especially for managing variables and scopes. We also have `generateDebugScopes` method, which generates the scopes, `scopeInsertionOrder` vector tracking the sequence of variable declaration in a function, and `scopeBuffer` vector storing produced scopes for printing. Furthermore, `localVars`, `tempVars`, and `tempVarCounter` are helper structures responsible for the register allocation and management, and structures like the basic `basicBlocks`, `basicBlocksOperations` string vectors, and `basicBlockCounter` supervise the control of the basic block.

3.2.4 Function Translation and Return Handling

The system processes full functions using `VisitFunctionDecl`, where C function signatures are converted into MIR function headers. Parameters are mapped to uniquely named registers like `_1`, and `_2` and the return value is always stored in register `_0`, while parameters are tracked at the same time. After traversing the function body, the visitor emits debug scopes and appends all collected basic blocks for execution flow. When a return statement is encountered, `VisitReturnStmt` ensures the value is stored in `_0`, and if possible, it simplifies the output by removing unnecessary variables which is an optimization that Rust's compiler also performs.

3.2.5 Expression and Control Flow Handling

Following functions and variables, translating expressions and control flow logic into the MIR format comes second most important. The Expression and Control Flow Handling component guarantees that every operation is shown in the output with correct semantics and structure. We have implemented various methods that handle binary operations, unary operations, functions, returns, loops, and conditionals. Binary operations are handled within `VisitBinaryOperator` method, which supports various arithmetic operations like `+`, `-`, `/`, `*` in a simple way like `a + b`, but also more complex chains such as `a + b + c`, guaranteeing overflow safe computation is

enforced using Rust-like `AddWithOverflow` and similar MIR intrinsics, combined with `assert` and `move` statements to maintain safety guarantees. The tool handles division and compound assignment operators `+=`, `-=`, `/=` and `*/`, and inserts runtime checks to catch division by zero or overflow, just like Rust would.

At last, control flow structures including `if` and `while` statements are handled by their corresponding visitor approaches. Dynamic basic blocks to model flow transitions, condition evaluations, and loop control. For example, `VisitWhileStmt` generates loop condition blocks, update blocks, and proper exits. These elements cooperate to guarantee that, at a lower level, Rust-style form fit for analysis or additional compilation, the translated output reflects the logic of the original C code.

3.2.6 Formatted Output Handling `(printf)`

This component of the system converts C printf statements into Rust-style MIR output. Instead of treating printf as a simple function call, the tool breaks it down into smaller steps that match how Rust internally formats and prints `println`! calls. It generates formatter objects for every variable, divides the format string into textual pieces, and promotes them as global constants. Then, just like actual Rust MIR, they are bundled together and passed to a special `_print(..)` call, ensuring that the output stays accurate, type safe, and loyal to the way Rust would handle the printing, even though the input was just a simple C printf.

3.2.7 Output Stream Management

The last component is in charge of producing the MIR-style output. Every C function is handled separately and generates its output separately as well. Every function the system generates a function header, debug information for its parameters, variable declarations, nested scopes, and matching basic blocks. These blocks include key operations such as assert, goto, and return, all formatted to match Rust's MIR style.

Every function's output is first created internally using a dedicated vector, `basicblocks`, which lets the system arrange the content before printing it. Ultimately,

the final output is just the aggregated view of the original C program formed by all previous function outputs, MIR-like.

Here is a generated block that manages a basic addition as output by the `VisitFuncDecl` approach:

```
    bb0: {
        _3 = AddWithOverflow(copy _1, copy _2);
        assert(!move (_3.1: bool), "attempt to compute `{} + {}`, which
would overflow", copy _1, copy _2) -> [success: bb1, unwind continue];
    }
```

3.3 Internal State and Structure Management

Throughout the translation process, the tool maintains several internal structures to keep track of variable names, scopes, basic block organization, and output ordering. These structures are essential for ensuring that the generated MIR code is consistent, correct, and semantically aligned with the original C source code.

3.3.1 Temporary Variables and Counters

One of the most important structures is the `tempVarCounter`, which tracks the number of temporary variables generated so far. This counter starts at the number of function parameters for each function and increments each time a new temporary variable is created during the traversal. This way, we are ensuring that temporary names like `_3`, `_4`, `_5`, and so on, are unique within the scope of each function.

```
    // Initialize the temp variable counter to the number of parameters
    tempVarCounter = Func->getNumParams();
```

Temporary variables are declared using MIR-style tuple types like `(i32, bool)` and stored in the `tempVars` vector. This vector helps prevent duplicate declarations and ensures that all temporaries are listed at the top of the function body in the final output.

3.3.2 Variable Mapping

To maintain consistent naming, the tool uses two main mappings:

- `paramMap`: stores the mapping between original parameter names and their MIR equivalents.
- `variableInfo`: a `map<string, VariableInfo>` that holds metadata for all declared variables. The `VariableInfo` struct includes details such as the variable's MIR name, initial value, type, usage status, return status, usage count, and more.

These mappings are cleared at the start of each new function and reused during traversal to resolve identifiers within expressions and statements.

3.3.3 Basic Block Management

The fundamental `basicBlockCounter` variable records the current block under creation. Every new block is stored in the `basicBlocksOperations` and `basicBlocks` vector and labelled with a MIR-style identifier (e.g., `bb2, bb3`).

The tool generates several blocks pushed to `basicBlocksOperations` in special cases including division overflow checks or loop updates. Once the function traversal is finished, this vector is flushed to guarantee proper sequence of emission of all control flow constructions.

3.3.4 Deferred Moves and Late Assignments

To mirror Rust MIR's separation between computation and value movement, the tool uses internal helpers like `moveLine`, which temporarily holds deferred assignment instructions. This lets operations like `AddWithOverflow` emit first, with the final assignment placed into the next basic block. Particularly in relation to conditional logic or operations that might cause panic, this design increases both correctness and readability. Likewise, for `while` loops, the `updateBlock` string keeps updates to the

loop control variable in a separate basic block, so guaranteeing accurate reevaluation of loop conditions following every iteration.

3.4 Execution Flow

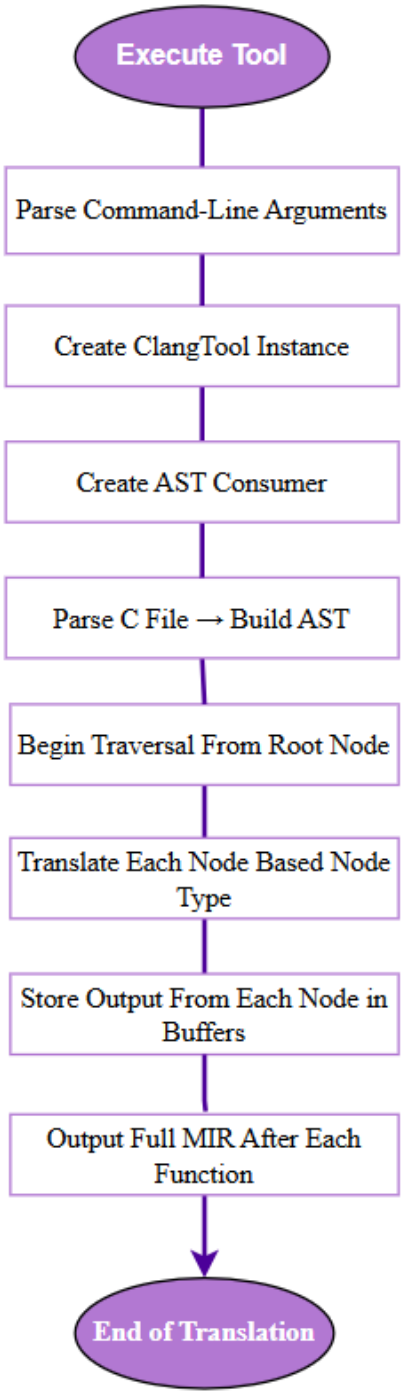
This subchapter is dedicated to explaining analytically the system's processing pipeline, specifically how the components mentioned in the previous subchapter interact dynamically during a real run of the tool. The tool is structured as a sequence of logically connected stages, each stage transforming the input and gradually refining raw C code into structured MIR-style output.

When the tool is executed, the first thing is to process the command line arguments and identify the source file to analyse using the `CommonOptionsParser` and create a `ClangTool` instance as mentioned previously. The control is therefore handed over to a custom frontend action, which links the internal logic to Clang's parsing pipeline, steps that prepare for the receiving and processing of the input source file. Clang then builds the Abstract Syntax Tree (AST) by parsing the C source file including all declarations, expressions, and control flow statements discovered in the original code.

A custom AST consumer is created once the frontend function is activated, which is notified when the translation unit is ready. At that point, the tool calls `TraverseDecl` on the root of the AST, which starts a recursive traversal of all nodes in the program. This trigger calls to specific `Visit*()` functions implemented by the visitor, such as `VisitFuncDecl`, `VisitBinaryOperator`, or `VisitReturnStmt`. Hence, this step marks the beginning of the actual analysis, as each node encountered is processed to extract its meaning and convert it to MIR-style output. As the traversal proceeds, the visitor logic performs translation based on its node type. For example, if the node processed is a function call, the tool does not shift its focus to the body of the called function. Instead, each function is processed independently in the order it appears in the source file, generating the corresponding MIR output. The body of the called function is only visited when the tool explicitly reaches its own declaration during traversal. This ensures that the source code is handled in a depth-first manner, preserving the logical order of the program. Rather than printing output immediately, the system stores it in various temporary buffers that are printed at the end of every function and cleared at the

beginning of every new one. This deferred approach allows the system to manage formatting, variable renaming, and proper ordering before the final output is written.

Overall, the execution flow of the system ensures that each part of the C program is visited, processed, and translated in an organized, depth-first manner. This structured pipeline allows the tool to generate complete, consistent MIR-style output while maintaining the semantic integrity of the original C source code, which is why this tool is needed.



3.5 Assumptions and Limitations

Several fundamental presumptions must be satisfied if the tool is to operate as expected and consistently:

i. Input C code is valid and compiles without errors: The tool expects the C code to be syntactically and semantically correct as if there are compilation errors, Clang will fail before the tool even starts traversing the AST.

ii. All code is self-contained in a single source file: All functions, variables, and types must be defined in the same .c file since no cross-file dependencies like `#include "otherfile.h"` that would require linking multiple translation units are supported.

iii. No indirect or dynamic function calls: Only direct function calls are supported, not function pointers or calls through function variables, resulting in an assumption from the tool that only direct function calls exist.

iv. Functions do not depend on global variables: The tool assumes functions rely on their parameters and local variables since global variable reads/writes are not specially tracked or translated into MIR.

v. No inline assembly or platform-specific extensions; the tool expects portable C code Clang can parse without custom extensions, hence non-standard language features or `asm(...)` are not handled.

vi. Simple expressions and supported operators: Arithmetic operations like `+`, `-`, `*`, `/` and simple comparisons like `==`, `!=`, `<`, `>` are expected, not complex ones like ternary operators `?:`

vii. Our tool expects that simple control flow structures which support basic control, partial `if` handling and `while` full handling are used in conventional, ordered manner.

Even with these assumptions in place, there are still some limitations to what the tool can currently do. These limitations reflect choices made during the design and development process, and they help define the boundaries of the system's functionality. The main limitations are outlined below:

i. Does not support global variables
ii. Does not support structures, unions, or pointers
iii. Does not support recursion or function call graph analysis
iv. Does not support multi-file compilation
v. Limited type handling
vi. Incomplete translation for control flow statements
vii. Incomplete optimization or simplification passes
viii. Limited error reporting
ix. Not an executable backend

Now that we have clearly defined our tool's architecture, execution flow, assumptions, and limitations, the next chapter will dive into how each major component of the tool was built and how each part of the translation process is implemented in practice.

Chapter 4

Implementation

4.1 AST Visitor and Translation Logic

The primary component of the translation works using Clang's RecursiveASTVisitor class to examine AST representations of parsed C programs. The MyASTVisitor visitor class focuses on key `Visit*` methods to convert language elements into Rust MIR intermediate representation. Rather than manually walking the tree node-by-node, Clang automatically dispatches the traversal to appropriate `Visit*` methods based on the type of the encountered AST node. The design of our tool allows modular and scalable handling of different constructs such as functions, expressions, and control flow statements.

Each overridden `Visit*` method focuses on:

- Analysing the specific AST node
- Emitting the equivalent MIR-like output
- Managing temporary variables, scope information, and basic blocks

A brief overview of the visitor methods implemented based on the node types, as well as some of the other implemented functions, is shown below:

Clang AST Node Type	Visitor Method	Purpose	Status
FunctionDecl	VisitFunctionDecl	Handle function signatures, parameters, and bodies	Implemented
VarDecl	VisitVarDecl	Handle variable declarations and type mapping	Implemented
DeclRefExpr	VisitDeclRefExpr	Tracks whether the variables are used and when	Implemented
BinaryOperator	VisitBinaryOperator	Handle binary operations (+, -, *, /) with overflow checking	Implemented
CallExpr	VisitCallExpr	Handle direct function calls Mention strings	Implemented
Helper	getExprName	Helper function to generate MIR-like operand strings from expressions	Implemented
Helper	generateDebugScopes	Build scope declarations for variables based on usage and insertion order	Implemented
Custom Logic	handlePrintf	Generate MIR-style formatted printing (_print(...)), dynamic formatting and promotion	Implemented
Custom Logic	emitPromotedConstant	Emit promoted format strings as MIR-style global constants for use in _print	Implemented
Custom Logic	hasSpecialFormatting	Detect formatting flags like %.2f, %05d in printf to enable new_v1_formatted calls	Implemented
ReturnStmt	VisitReturnStmt	Handle return statements and return value movement	Implemented
IfStmt	VisitIfStmt	Handle if-else conditional branches	Partial
WhileStmt	VisitWhileStmt	Handle while loops	Implemented
UnaryOperator	VisitUnaryOperator	Log and recognize unary operations	Partial

Unimplemented or partially implemented visitor methods are discussed later in the Future Work chapter.

With the overall AST traversal strategy and the key visitor methods outlined, we can now dive deeper into how some of the most critical components of the translation are implemented. In the next sections, we will explore the logic behind handling function declarations, expressions, control flow statements, and special cases, shedding light on the decisions made to ensure an accurate and robust translation into MIR-like output.

4.2 Handling Functions and Parameters

C program's functions serve as fundamental programming elements for the accuracy and reliability of the MIR output generated, since C programs are consisted of multiple functions. The `VisitFunctionDecl` method is responsible for this translation, handling everything from function signatures to local variables and basic blocks. In this section, we will explain how the tool processes a function, step-by-step, and the motivation behind every design choice.

The method uses a `FunctionDecl` named `*Func` as a parameter and first aims to differentiate between a full function definition with a body and a simple function declaration such as a prototype `int foo();`. Since we are only interested in functions that contain code, the visitor skips any declarations without a body:

```
    if (!Func->isThisDeclarationADefinition())
      return true; // Skip function declarations without a body
```

The next step is to extract the function's name using the `getNameAsString()` method that is inherited from Clang's `NameDecl` class, which provides a convenient way to extract the identifier name of any named declaration, including functions. Then, we move on to clearing all internal structures used within our tool, ensuring that each new function traversal starts with a clean state.

Before generating the function's code, we initialize a counter, `tempVarCounter`, which is responsible for tracking how many variables, including parameters and temporary ones exist. This counter starts at the number of function parameters thus any additional variables created later have the correct parameter names.

```
    // Initialize the temp variable counter to the number of parameters
    tempVarCounter = Func->getNumParams();
```

Next, the return type of the function is extracted and mapped into a Rust compatible type. This is necessary because MIR expects Rust typing conventions:

```
    // Translate C types to Rust types
    if (returnType == "void")
      returnType = "()";
    else if (returnType == "int")
      returnType = "i32";
    else if (returnType == "unsigned int")
      returnType = "u32";
    else if (returnType == "float")
      returnType = "f32";
    else if (returnType == "double")
      returnType = "f64";
    else if (returnType == "char")
      returnType = "char";
    else if (returnType == "bool")
      returnType = "bool";
    else if (returnType == "char *" || returnType == "const char *")
      returnType = "&str";

    // Special case for main
    if (Func->getNameAsString() == "main")
      returnType = "()";
```

Additionally, the `main` function is treated as a special case, always returning `()`, the Rust equivalent of `void`.

We continue with printing the function's signature and parameters in the Rust MIR syntax. Each parameter is automatically given a name like _1 or _2, mapped to its Rust type and save it into an unordered map called paramMap for later use:

```
    // Print function signature
    llvm::outs() << "fn " << Func->getNameAsString() << "(";

    for (unsigned i = 0; i < Func->getNumParams(); ++i)
    {
      if (i > 0)
        llvm::outs() << ", ";
```

```
      std::string paramType = Func->getParamDecl(i)-
>getType().getAsString();
      if (paramType == "int")
        paramType = "i32";
      else if (paramType == "unsigned int")
        paramType = "u32";
      else if (paramType == "float")
        paramType = "f32";
      else if (paramType == "double")
        paramType = "f64";
      else if (paramType == "char *" || paramType == "const char *")
        paramType = "&str";
      else if (paramType == "char")
        paramType = "char";
      else if (paramType == "bool")
        paramType = "bool";
      llvm::outs() << "_" << (i + 1) << ": " << paramType;

      std::string paramName = Func->getParamDecl(i)->getNameAsString();

      // Store mapping (e.g., "a" -> "_1", "b" -> "_2")
      paramMap[paramName] = "_" + std::to_string(i + 1);
    }
```

Following the function signature, the translator emits debug lines for each function parameter, if any, mimicking the way Rust's compiler treats parameters separately from the internal temporaries. This step is important for tools like debuggers or static analyzers that use source-level information.

```
    // Ensure we only print debug statements IF the function has
parameters
    if (tempVarCounter > 0)
    {
      for (unsigned i = 0; i < tempVarCounter; ++i)
      {
        llvm::outs() << "    debug "
                     << Func->getParamDecl(i)->getNameAsString()
                     << " => _" << (i + 1) << ";\n";
      }
    }
```

Before continuing with the traversal of the function's body, we create the return variable `_0`, which holds the return value for the function, mirroring the behaviour of Rust's MIR.

```
    // Printing the return variable
    llvm::outs() << "    let mut _0: " << returnType << ";\n";
```

The tool moves on to processing the actual content, the statements and expressions that compose the body. First, it determines if the function has a body and, if so, calls it to traverse it. During this traversal, each child node is handled by the corresponding `Visit*` method, whether it is a statement like a return statement, while statement, if statement, call expression, or a binary or unary operation.

```
    if (Stmt *Body = Func->getBody())
    {
      TraverseStmt(Body); // Process the body of the function.
    }
```

After the traversal is completed, all local variables encountered or created within the function are printed, followed by the generation and printing of debug scopes and the printing of all basic blocks stored in the `basicBlocks` vector. The function concludes by resetting some final vectors.

```
    // Output all basic blocks with dynamically generated labels.
    for (size_t i = 0; i < basicBlocks.size(); i++)
    {
      llvm::outs() << basicBlocks[i];
    }
```

4.3 Handling Binary Operations

Binary operations are a core component of computational logic within most C programs and one of the first components we implemented. Their presence, from simple arithmetic expressions to conditional checks within loops and control structures, requires precise translation into Rust's Mid-level Intermediate Representation (MIR) to preserve semantic accuracy.

In this section, we will take a closer look at how the tool processes binary operations by implementing the `VisitBinaryOperator` method. We walk through how it handles type resolution, manages temporary variables, performs overflow checks, and

dynamically generates basic blocks in a style consistent with Rust's MIR. Additionally, we discuss a special case involving binary operations within while loop conditions, which requires careful handling to ensure correct placement and flow.

First step in the `VisitBinaryOperator` method is to ensure that the tool skips both system headers and relational operators, which are handled only as part of if statements inside `VisitIfStmt` method.

```
    // Ignore system header operations
   if (Context->getSourceManager().isInSystemHeader(BinOp->getExprLoc()))
      return true;

    std::string Op = BinOp->getOpcodeStr().str();
    std::string rustOp;

    // Skip relational operators—they should be handled inside
`VisitIfStmt()`
    if (Op == ">" || Op == "<" || Op == "==" || Op == "!=")
    {
      return true;
    }
```

Next step is to retrieve the operand's type and convert it to appropriate MIR-compatible type.

C Operator	MIR Equivalent
+, +=	AddWithOverflow
-, -=	SubWithOverflow
*, *=	MulWithOverflow
/, /=	DivWithOverflow

The visitor proceeds to analyse both sides of the expression, unwrapping any implicit casts to reveal the underlying variables or literals. Then, operand names are resolved using `getNameInfo().getAsString()`. If the variable corresponds to a function parameter, it is then replaced with its MIR name from paramMap. Otherwise, the resolved variable name is used directly.

Before performing general operand analysis and MIR block generation, the visitor first checks whether the binary operation is occurring inside a while loop. If so, it further verifies whether the variable being modified by the operation matches the loop control variable previously extracted from the condition. This selective filtering ensures that only relevant operations such as `i /= 1` in a loop over `i` are handled specially. When such a match is found, the tool generates overflow-aware MIR instructions tailored for loop-safe arithmetic and appends a `goto` instruction that re-evaluates the loop condition. Division operations receive additional safeguards, including checks for division by zero and signed overflow like `i32::MIN / -1`, ensuring MIR semantics remain correct and robust during loop iteration.

How the relevant basic blocks are structured:

```
            blockLabel += moveLine;
            moveLine = "";

            // Basic block for division-by-zero check
            blockLabel += "        " + divZeroTemp + " = Eq(" +
formatOperand(rhsVar) + ", const 0_i32);\n";
            blockLabel += "        assert(!move " + divZeroTemp + ",
\"attempt to divide `{}` by zero\", " + formatOperand(lhsVar) +
                          ") -> [success: bb" + std::to_string(nextBB) +
", unwind continue];\n";
            blockLabel += "    }\n";

            basicBlocksOperations.push_back(blockLabel);

            blockLabel = "\n    bb" + std::to_string(nextBB) + ": {\n";
            nextBB = basicBlockCounter++;

            // Overflow check (-1 divisor case)
            blockLabel += "        " + negTemp + " = Eq(" +
formatOperand(rhsVar) + ", const -1_i32);\n";
            blockLabel += "        " + minTemp + " = Eq(" +
formatOperand(lhsVar) + ", const i32::MIN);\n";
            blockLabel += "        " + ovfTemp + " = BitAnd(move " +
negTemp + ", move " + minTemp + ");\n";
            blockLabel += "        assert(!move " + ovfTemp + ",
\"attempt to compute `{} / {}`, which would overflow\", " +
formatOperand(lhsVar)
                          + ", " + formatOperand(rhsVar) + ") ->
[success: bb" + std::to_string(nextBB) + ", unwind continue];\n";
            blockLabel += "    }\n";
```

```
            basicBlocksOperations.push_back(blockLabel);

            blockLabel = "\n    bb" + std::to_string(nextBB) + ": {\n";

            // Division operation
            blockLabel += "        " + lhsVar + " = Div(" +
formatOperand(lhsVar) + ", " + formatOperand(rhsVar) + ");\n";
            blockLabel += "        goto -> bb" + std::to_string(condBB) +
";\n";
            blockLabel += "    }\n";

            basicBlocksOperations.push_back(blockLabel);
```

For all other binary operations outside of loops, the tool follows a general pattern: it first identifies the target variable receiving the result, determines the correct MIR intrinsic for the operation, and creates a new temporary variable to store the result along with an overflow flag. It then generates a basic block containing the computation and emits an `assert!` to ensure the operation did not overflow. Division operations are handled slightly differently, as we previously explained above where we talked about handling operations on the while loop control variable. Instead of assigning the result immediately, the tool stores it in the helper string called `moveLine` that we talked about in chapter 3.

4.4 Control Flow Statements

Implementing various control flow statements is undoubtedly one of the most important and challenging parts of our tool's implementation. Control flow statements can have various forms and can combine multiple operations, expressions and nested statements. For example, one while loop may contain several binary operations, function calls, printf statements and conditional branches, all of which must interact flawlessly. That was the primary goal for our tool, to combine every operation together simultaneously, and achieve a near-perfect translation.

4.4.1 While Loops

To handle while loops, we implemented `VisitWhileStmt` method. We firstly extract the condition of the loop and unwrap nested `ImplicitCastExpr` nodes, if present, to access the actual loop control variable and store it in `whileLoopVar` global string. We also set the flag `insideWhileLoop` to true, which will be later used in `VisitBinaryOperator` to determine whether an operation is an operation on the condition variable of the loop. Appropriate error messages are printed, and the loop is skipped when control variable can not be resolved.

```
    // Extract Condition
    std::string loopCond;
    if (While->getCond())
    {
      llvm::raw_string_ostream rso(loopCond);
      While->getCond()->printPretty(rso, nullptr,
PrintingPolicy(LangOptions()));
    }
```

After extracting the condition, the tool creates temporary variables for the loop variable and the evaluated condition, with their types translated to the appropriate MIR ones. Basic blocks are then generated for the initialization of the loop variable and evaluation of the condition. In the condition block, the loop variable is copied, and a MIR-style comparison operation like `Lt`, `Gt` or `Eq` is applied using the operands extracted from the original C expression. This comparison result is stored in `condVar`, which is then used in a switchInt statement to direct control flow, since the tool translates integer condition variables. If the condition is false, the loop exits. Otherwise, control transfers to the block where the loop body begins execution. Next step is to traverse the body of the while so every operation inside the while loop can be processed and correctly placed within the while scope.

```
    // Now lets traverse the rest of the while body
    TraverseStmt(While->getBody());
```

Finally, after the traversal all basic blocks are pushed inside the basicBlocks vector, and the loop exists by transferring the control to a newly created exit block.

4.4.2 If / Else if / Else

The VisitIfStmt approach has partially applied support for conditional branching. The aim is to translate C if, else if, and else blocks into distinct basic blocks with straightforward MIR-style control flow. The approach can currently extract simple boolean conditions and create a switchInt statement to branch execution between a then and an else block.

The arrangement of the emitted blocks is not yet totally accurate, though. Many times, nested conditions, several else if chains, and return-value flow between branches go unpackled as expected. The resulting MIR might not faithfully reflect the semantics of the original C code even while switchInt and block labels are produced.

This part aims to record the design and early implementation phases. Part of the intended future work is full support for appropriately nested if/else blocks. This function is still experimental until then; accurate translation cannot depend on it.

4.5 Handling of Printing

A major milestone in this thesis was the ability to correctly interpret and convert `printf` statements in C code into semantically equivalent Rust `println!` invocations, expressed in MIR. Since formatted printing is a critical component in real world C programs, our tool was designed to support both simple printing and advanced formatted output with variable interpolation, consistent with how Rust MIR handles such constructs.

4.5.1 Basic String Printing

The first step in successfully translating print statements was to support basic printf calls that print constant format strings and the following format specifiers: `%d`, `%i`, `%u`, `%f`, `%e`, `%E`, `%x`, `%X`, `%c`, and `%s`, without any width or precision modifiers. These were translated into calls to Rust's `_print()` macro using `Arguments::new_v1()` and a

promoted constant slice representing the format string. For example, the following C line:

```
printf("Unsigned: %u | Hex: %x | UpperHex: %X\n", num, num, num);
```

would be translated into MIR-style calls using `new_display`, `new_lower_hex`, and `new_upper_hex` respectively, along with a `&[&str; N]` constant representing the static string segments.

4.5.2 Support for Width, Precision, and Format Specifiers

Once basic specifiers were working, the tool was extended to correctly handle optional width and precision modifiers like `%05d`, `%.2f`, `%8.3e`. These are parsed using a regular expression that captures the width, precision, and type specifier. When any modifier is detected, the tool emits a call to `Arguments::new_v1_formatted`, including a constructed array of `Placeholder` objects with the correct alignment, padding, flags, and counts. This approach mirrors how Rust MIR models format output internally and ensures consistent behavior even for more complex formatting instructions.

4.5.3 Type Mapping and Format Selection

Each printf argument is mapped to its Rust equivalent by inspecting the original C type through Clang's AST. This mapping ensures that format specifiers like `%d`, `%f`, or `%x` use the correct MIR constructors and Rust types.

Common mappings include:

- `%d` / `%i` → `i32` or `i64` with `new_display`
- `%f`, `%.2f` → `f32` or `f64`, optionally with `new_lower_exp`
- `%x`, `%X` → `u32` / `u64` with `new_lower_hex` / `new_upper_hex`
- `%s` → `&str` from char arrays or string literals

If a format specifier does not match the argument's inferred type, the tool falls back to a safe default or emits an error, minimizing undefined behaviour in the resulting MIR.

4.5.4 Static Segment Promotion and Formatting Arrays

Mirroring Rust MIR's treatment of format strings, the tool divides the format string into stationary segments and encodes them into a `&[&str; N]` constant array.
Corresponding Argument and Placeholder arrays are created to hold references to the runtime values and their formatting specifications. Depending on whether formatting modifiers exist, these arrays are then forwarded to either `Arguments::new_v1` or `Arguments::new_v1_formatted`.
Every array is stored in a separate temporary variable, and slice coercions are used as needed. While maintaining the layout and intent of the original C `printf`, this design guarantees type-safe, Rust-style formatting behaviour.

4.5.5 Deferred Argument Emission

To ensure that our tool has correct evaluation order and MIR compliance, argument moves like `move _x` are deferred into a separate basic block using the `moveLine` buffer. The tool first emits all reference and `Argument::new_*` expressions, deferring actual `move` operations until after those constructors complete, using the mechanism described earlier in section 3.3.4: Deferred Moves and Late Assignments.

4.6 Tool Setup and Execution

By the end of this subchapter, the user will be able to configure, build, and run the tool. The chapter contains details about the operating system used during the development of the tool, the necessary installations and dependencies, and the build and execution process.

4.6.1 Dependencies

The tool was developed and tested on Ubuntu 22.04.5 LTS and uses the Clang and LLVM components. The following packages need to be installed:

```
sudo apt install llvm-dev clang libclang-dev llvm
```

The additional dependencies are also necessary for building and compiling the tool:

```
sudo apt install -y \
    build-essential \
    cmake \
    ninja-build \
    python3 \
    python3-pip \
    git \ libedit-dev \
    libxml2-dev \
    zlib1g-dev \
    libncurses5-dev \
    libssl-dev
```

4.6.2 Building the Tool

We need to create a directory like '`c_to_mir`' and add the `CMakeLists.txt` and `ast_traversal.cpp` files inside the new directory. Then, create a subdirectory for building the project using the following commands:

```
mkdir build
cd build
```

We proceed on building the files and compiling the tool:

```
cmake ..
make
```

If the building and compiling were successful, it will produce an executable called `astTraverse` inside the `build/` directory.

4.6.3 Execution

To run the tool on a C file and produce the MIR-style output you need to type the following command in a terminal inside the `build/` directory.

```
./astTraverse path/to/file.c
```

The tool will parse the C source file, traverse its Abstract Syntax Tree (AST), and print the corresponding MIR-style code to the terminal. This includes function declarations, variable declarations, scopes, and control flow blocks.

Chapter 5

Evaluation

5.1 Testing Strategy

We assessed our tool against a well-chosen collection of C programs covering many language aspects including function calls, arithmetic operations, loops, if statements, and formatted printing using printf. Our aim was to verify that the output generated matches the structure and semantics of the Rust compiler's own MIR as closely as possible. For comparison, we generated equivalent Rust code for each C example, compiled it with `-emit=mir`, and manually inspected the MIR output to compare with the tool's translation.

5.2 Selected Test Cases and Results

To evaluate the correctness and completeness of the translation process, we designed and executed 23 test programs written in C. These programs were carefully selected to cover a wide range of language features, including arithmetic operations, control flow structures, function call, return statements, and formatted output using `printf`.

For each test case, the tool was executed to produce a Rust-style MIR output. The resulting MIR was compared against MIR generated by the Rust compiler with an equivalent Rust version of the C code.

The table below lists every test case, feature tested, and observed output result. After that, particular test cases are thoroughly shown together with the original C code, matching MIR output, and a justification of the correctness.

C Program	C Program Description	Features Tested
add.c	Add two integers with overflow check	AddWithOverflow
minus.c	Subtract two integers with overflow check	SubWithOverflow
mul.c	Multiply two integers	MulWithOverflow
divide.c	Divide two integers	DivWithOverflow
call_test.c	Simple function call (square)	Function call
complex_return_chain.c	Nested call with intermediate computations	Chained call, Add+Mul
emptyfunction.c	Function with no body	Void function
expirement.c	Temp variable reused before return	Binary ops, reuse
expirementdiv.c	Division followed by multiplication	DivWithOverflow, chained op
expirementused.c	Add + print before return	Print, temp var reuse
fmt_test.c	Multiple format specifiers, precision, padding	Formatted printing
format_char_string.c	Print character and string	Char & string formatting
format_hex_unsigned.c	Print unsigned and hex representations	u, x, X formatting
format_scientific_pointer.c	Scientific float formatting	%e specifiers
multiple_addition.c	Chained addition in one line	Flattened binary ops
print_complex.c	Combined printf with mixed types	Advanced printf
printing.c	Add + print result	Print + AddWithOverflow
printing_multi.c	Add + Mul with printf	Chained ops with print
test.c	Useless assignment followed by return	Unused vars
while.c	Basic while loop with increment	While loop
whilediv.c	While loop with division	Loop + DivWithOverflow
whilereturn.c	While loop with return of counter	Loop + return
full_power_test.c	Function call, addition, multiplication, loop, printf, return	AddWithOverflow, MulWithOverflow, While loop, Function call, Print

Test Case 1: Formatted Output and Arithmetic `(fmt_test.c)`

C source code:

```c
#include <stdio.h>

int report(int a, int b, float f) {
    int sum     = a + b;
    int product = a * b;

    // simple integer print
    printf("Sum: %d, Product: %d\n", sum, product);
    printf("Sum padded: %05d, Float precise: %8.3f\n", sum, f);

    return sum + product;
}

int main() {
    report(5, 7, 3.14159f);
    return 0;
}
```

Generated MIR code:

```
fn report(_1: i32, _2: i32, _3: f32) -> i32 {
    debug a => _1;
    debug b => _2;
    debug f => _3;
    let mut _0: i32;
    let mut _4: i32;
    let mut _5: (i32, bool);
    let mut _6: i32;
    let mut _7: (i32, bool);
    let _8: ();
    let mut _9: std::fmt::Arguments<'_>;
    let _10: &[&str; 3];
    let _11: &[core::fmt::rt::Argument<'_>; 2];
    let _12: [core::fmt::rt::Argument<'_>; 2];
    let mut _13: core::fmt::rt::Argument<'_>;
    let _14: &i32;
    let mut _15: core::fmt::rt::Argument<'_>;
    let _16: &i32;
    let _17: ();
    let mut _18: std::fmt::Arguments<'_>;
```

```
    let mut _19: &[&str];
    let _20: &[&str; 3];
    let mut _21: &[core::fmt::rt::Argument<'_>];
    let _22: &[core::fmt::rt::Argument<'_>; 2];
    let _23: [core::fmt::rt::Argument<'_>; 2];
    let mut _24: core::fmt::rt::Argument<'_>;
    let _25: &i32;
    let mut _26: core::fmt::rt::Argument<'_>;
    let _27: &f32;
    let mut _28: &[core::fmt::rt::Placeholder];
    let _29: &[core::fmt::rt::Placeholder; 2];
    let _30: [core::fmt::rt::Placeholder; 2];
    let mut _31: core::fmt::rt::Placeholder;
    let mut _32: core::fmt::rt::Alignment;
    let mut _33: core::fmt::rt::Count;
    let mut _34: core::fmt::rt::Count;
    let mut _35: core::fmt::rt::Placeholder;
    let mut _36: core::fmt::rt::Alignment;
    let mut _37: core::fmt::rt::Count;
    let mut _38: core::fmt::rt::Count;
    let mut _39: core::fmt::rt::UnsafeArg;
    let mut _40: (i32, bool);
    scope 1 {
        debug sum => _4;
        scope 2 {
            debug product => _6;
        }
    }

    bb0: {
        _5 = AddWithOverflow(copy _1, copy _2);
        assert(!move (_5.1: bool), "attempt to compute `{} + {}`, which
would overflow", copy _1, copy _2) -> [success: bb1, unwind continue];
    }

    bb1: {
        _4 = move (_5.0: i32);
        _7 = MulWithOverflow(copy _1, copy _2);
        assert(!move (_7.1: bool), "attempt to compute `{} * {}`, which
would overflow", copy _1, copy _2) -> [success: bb2, unwind continue];
    }

    bb2: {
        _6 = move (_7.0: i32);
        _10 = const report::promoted[1];
        _14 = &_4;
        _13 = core::fmt::rt::Argument::<'_>::new_display::<i32>(copy _14)
-> [return: bb3, unwind continue];
```

```
    }

    bb3: {
        _16 = &_6;
        _15 = core::fmt::rt::Argument::<'_>::new_display::<i32>(copy _16)
-> [return: bb4, unwind continue];
    }

    bb4: {
        _12 = [move _13, move _15];
        _11 = &_12;
        _9 = Arguments::<'_>::new_v1::<3, 2>(copy _10, copy _11) ->
[return: bb5, unwind continue];
    }

    bb5: {
        _8 = _print(move _9) -> [return: bb6, unwind continue];
    }

    bb6: {
        _20 = const report::promoted[0];
        _19 = copy _20 as &[&str] (PointerCoercion(Unsize));
        _25 = &_4;
        _24 = core::fmt::rt::Argument::<'_>::new_display::<i32>(copy _25)
-> [return: bb7, unwind continue];
    }

    bb7: {
        _27 = &_3;
        _26 = core::fmt::rt::Argument::<'_>::new_display::<f32>(copy _27)
-> [return: bb8, unwind continue];
    }

    bb8: {
        _23 = [move _24, move _26];
        _22 = &_23;
        _21 = copy _22 as &[core::fmt::rt::Argument<'_>]
(PointerCoercion(Unsize));
        _32 = core::fmt::rt::Alignment::Unknown;
        _33 = core::fmt::rt::Count::Implied;
        _34 = core::fmt::rt::Count::Is(const 5_usize);
        _31 = core::fmt::rt::Placeholder::new(const 0_usize, const ' ',
move _32, const 8_u32, move _33, move _34) -> [return: bb9, unwind
continue];
    }

    bb9: {
        _36 = core::fmt::rt::Alignment::Unknown;
```

```
        _37 = core::fmt::rt::Count::Is(const 3_usize);
        _38 = core::fmt::rt::Count::Is(const 8_usize);
        _35 = core::fmt::rt::Placeholder::new(const 1_usize, const ' ',
move _36, const 0_u32, move _37, move _38) -> [return: bb10, unwind
continue];
    }

    bb10: {
        _30 = [move _31, move _35];
        _29 = &_30;
        _28 = copy _29 as &[core::fmt::rt::Placeholder]
(PointerCoercion(Unsize));
        _39 = core::fmt::rt::UnsafeArg::new() -> [return: bb11, unwind
continue];
    }

    bb11: {
        _18 = Arguments::<'_>::new_v1_formatted(move _19, move _21, move
_28, const core::fmt::rt::UnsafeArg {{ _private: () }}) -> [return: bb12,
unwind continue];
    }

    bb12: {
        _17 = _print(move _18) -> [return: bb13, unwind continue];
    }

    bb13: {
        _40 = AddWithOverflow(copy _4, copy _6);
        assert(!move (_40.1: bool), "attempt to compute `{} + {}`, which
would overflow", copy _4, copy _6) -> [success: bb14, unwind continue];
    }

    bb14: {
        _0 = move (_40.0: i32);
        return;
    }
}

const report::promoted[0]: &[&str; 3] = {
    let mut _0: &[&str; 3];
    let mut _1: [&str; 3];

    bb0: {
        _1 = [const "Sum padded: ", const ", Float precise: ", const
"\n"];
        _0 = &_1;
        return;
    }
```

```
}

const report::promoted[1]: &[&str; 3] = {
    let mut _0: &[&str; 3];
    let mut _1: [&str; 3];

    bb0: {
        _1 = [const "Sum: ", const ", Product: ", const "\n"];
        _0 = &_1;
        return;
    }
}

fn main() -> () {
    let mut _0: ();
    let _1: i32;

    bb0: {
        _1 = report(const 5_i32, const 7_i32, const 3.1415901f32) ->
[return: bb1, unwind continue];
    }

    bb1: {
        return;
    }
}
```

This example demonstrates the full capability of our tool's formatted output generation, including dynamic format string promotion, padding and precision handling, use of Placeholder objects, and safe arithmetic with overflow assertions. The structure, control flow, and formatting match the MIR emitted by the Rust compiler.

The only difference lies in variable scoping: the Rust compiler moves some variables into nested scope {} blocks based on usage to optimize lifetime management and memory. Our tool currently emits all let declarations at the function's top level since we did not handle that optimization.

Test Case 2: Nested Functions, Loops, and Print `(full_power_test.c)`

C source code:

```c
#include <stdio.h>

int multiply(int a, int b) {
    return a * b;
}

int square_sum_loop(int x, int y) {
    int sum = x + y;
    int result = multiply(sum, sum);

    int counter = 0;
    while (counter < result) {
        counter += 1;
    }

    printf("Final result is: %d, Loop count: %d\n", result, counter);
    return result;
}

int main() {
    int output = square_sum_loop(2, 3);
    return 0;
}
```

Generated MIR code:

```
fn multiply(_1: i32, _2: i32) -> i32 {
    debug a => _1;
    debug b => _2;
    let mut _0: i32;
    let mut _3: (i32, bool);

    bb0: {
        _3 = MulWithOverflow(copy _1, copy _2);
        assert(!move (_3.1: bool), "attempt to compute `{} * {}`, which
would overflow", copy _1, copy _2) -> [success: bb1, unwind continue];
    }

    bb1: {
        _0 = move (_3.0: i32);
        return;
```

```
    }
}

fn square_sum_loop(_1: i32, _2: i32) -> i32 {
    debug x => _1;
    debug y => _2;
    let mut _0: i32;
    let mut _3: i32;
    let mut _4: (i32, bool);
    let mut _5: i32;
    let mut _6: i32;
    let mut _7: bool;
    let mut _8: i32;
    let mut _9: (i32, bool);
    let _10: ();
    let mut _11: std::fmt::Arguments<'_>;
    let _12: &[&str; 3];
    let _13: &[core::fmt::rt::Argument<'_>; 2];
    let _14: [core::fmt::rt::Argument<'_>; 2];
    let mut _15: core::fmt::rt::Argument<'_>;
    let _16: &i32;
    let mut _17: core::fmt::rt::Argument<'_>;
    let _18: &i32;
    scope 1 {
        debug sum => _3;
        scope 2 {
            debug result => _5;
            scope 3 {
                debug counter => const 0_i32;
            }
        }
    }

    bb0: {
        _4 = AddWithOverflow(copy _1, copy _2);
        assert(!move (_4.1: bool), "attempt to compute `{} + {}`, which
would overflow", copy _1, copy _2) -> [success: bb1, unwind continue];
    }

    bb1: {
        _3 = move (_4.0: i32);
        _5 = multiply(copy _3, copy _3) -> [return: bb2, unwind
continue];
    }

    bb2: {
        _6 = const 0_i32;
        goto -> bb3;
```

```
    }

    bb3: {
        _8 = copy _6;
        _7 = Lt(move _8, copy _5);
        switchInt(move _7) -> [0: bb6, otherwise: bb4];
    }

    bb4: {
        _9 = AddWithOverflow(copy _6, const 1_i32);
        assert(!move (_9.1: bool), "attempt to compute `{} + {}`, which
would overflow", copy _6, const 1_i32) -> [success: bb5, unwind
continue];
    }

    bb5: {
        _6 = move (_9.0: i32);
        goto -> bb3;
    }

    bb6: {
        _12 = const square_sum_loop::promoted[0];
        _16 = &_5;
        _15 = core::fmt::rt::Argument::<'_>::new_display::<i32>(copy _16)
-> [return: bb7, unwind continue];
    }

    bb7: {
        _18 = &_6;
        _17 = core::fmt::rt::Argument::<'_>::new_display::<i32>(copy _18)
-> [return: bb8, unwind continue];
    }

    bb8: {
        _14 = [move _15, move _17];
        _13 = &_14;
        _11 = Arguments::<'_>::new_v1::<3, 2>(copy _12, copy _13) ->
[return: bb9, unwind continue];
    }

    bb9: {
        _10 = _print(move _11) -> [return: bb10, unwind continue];
    }

    bb10: {
        _0 = copy _5;
        return;
    }
```

```
}

const square_sum_loop::promoted[0]: &[&str; 3] = {
    let mut _0: &[&str; 3];
    let mut _1: [&str; 3];

    bb0: {
        _1 = [const "Final result is: ", const ", Loop count: ", const
"\n"];
        _0 = &_1;
        return;
    }
}

fn main() -> () {
    let mut _0: ();
    let mut _1: i32;
    scope 1 {
        debug output => _1;
    }

    bb0: {
        _1 = square_sum_loop(const 2_i32, const 3_i32) -> [return: bb1,
unwind continue];
    }

    bb1: {
        return;
    }
}
```

This example shows several key translators' capabilities: nested function calls, arithmetic with overflow checks, a while-style loop applying switchInt, and formatted output using Arguments::new_v1. The tool dynamically generates print arguments depending on value type and position, correctly emitting promoted string constants.

Like the previous example, the tool's output matches the actual Rust MIR almost exactly. The only noticeable difference is the placement of variable declarations, that some are not promoted into scopes. Rust MIR generates let declarations at the top of each function, while it nests variables including result and counter within inner scopes for memory and lifetime optimization, a future improvement area of our tool.

Conclusion

These case studies show that the tool generates MIR structures nearly identical to the actual Rust compiler, including safety checks, formatted printing, and value flow. Structural differences are minimal and mostly related to variable scoping. These findings show that the tool can produce precisely MIR-compliant output from actual C programs.

Chapter 6

Future Work

Time restrictions allow several areas for improvement even though the tool presently produces consistent and ordered MIR-style output from C source code. The remaining tasks relate to supporting language constructs, improving semantic accuracy, mimicking the compiler's optimizations, and preparing the tool for broader usability and maintainability.

6.1 Support for Additional Language Constructs

To increase the translator's compatibility with real world C programs there are several important constructs that need to be implemented. Those include:

- **Variable Initialization:** The tool successfully handles initializations from operations like `int x = a + b;`, expressions, and function calls, but not simple constant initializations like `int a = 5;` since Rust's compiler has optimizations that sometimes handle those initializations inside the scopes.
- **Unary Operators:** Support for unary negation, logical negation `!x`, bitwise complement `~x`, and increment/decrement operations `++x`, `x--` is required.
- **For loops:** We do not have any kind of for statement handling since we focused on fully implementing while loops. Full translation into equivalent MIR loop constructs is necessary, including support for loop initializers, condition checks, and iteration checks.
- **Switch Statements:** Full implementation of `VisitSwitchStmt` function is needed, including actual case-by-case branching logic, break handling, and default case support.
- **If Statements:** The `VisitIfStmt` logic is partially implemented. It needs full dynamic construction of conditional blocks, else if chains, final else fallbacks, and correct placement of return and goto statements.

6.2 Advanced Features: Structs, Arrays, and Casts

Many advanced C features still lack support and clearly point the way to where future development should go:

- Structs and Members: The tool does not yet translate struct fields or member access expressions like `obj.field` or `ptr->field` since `VisirRecorDecl` is not yet implemented.
- Arrays: Currently ignores expressions like `arr[i]` or pointer arithmetic with array indices are currently ignored. There has to be added proper memory indexing logic.
- Pointers: Currently, pointer variables are excluded by design. Future versions may include safe representations or modeled dereferencing `(*ptr)` and address-of `(&x)` operations where possible.
- Explicit C-style Casts: Common in C, expressions such `(int)x`, are not yet handled by the translator. Handling these correctly via `VisitCStyleCastExpr` is necessary for semantic completeness.

6.3 Compiler Optimization Fidelity

A major long-term goal is to mimic all MIR-level optimizations that the Rust compiler performs:

- Elimination of unused variables (dead code removal).
- Simplification of return paths by collapsing intermediate temporaries when they are only used once.
- Inlining and scope compression for short-lived temporaries.
- Efficient move semantics and deferral of assignments (already partially implemented using moveLine).
- Promotion of constants for formatting output, and reuse of identical formatting data.

This would ensure that the generated MIR not only looks correct but also behaves and performs like actual compiler-emitted MIR.

6.4 Evaluation Method

The next step is to evaluate the tool's output using Miri. Miri is a useful tool for interpreting Rust MIR and looking for undefined behavior, memory errors, and other safety issues. However, Miri does not accept MIR code directly, instead, it executes Rust source code by interpreting the MIR generated internally by `rustc`.
We identified two potential strategies to evaluate our tool's MIR-style output with Miri.

1. **Reconstructing MIR as Rust Code**
   One theoretical part would be to reconstruct Rust from our MIR code. This reconstructed Rust code could then be compiled with `rustc` and executed using Miri, allowing indirect validation of our output. However, this method would involve significant reverse-engineering and ultimately double our tool's workload which will result in significantly increase the tool's complexity. Nevertheless, this approach remains viable for future extensions.

2. **Forking rustc to modify the MIR generation phase**
   A second, more invasive method is to fork the official `rust-lang/rust` and modify the MIR generation stage within the compiler itself to inject or replace MIR with the output of our tool. While this would technically allow direct evaluation with Miri, such a solution is highly complex, time-consuming, and requires deep integration into the compiler's internals. Nonetheless, it represents a compelling future research direction for those aiming to integrate C-to-MIR translation more deeply into the Rust toolchain.

This subchapter purposefully features a brief review of these Miri-related restrictions and possible approaches. For deeper integration and validation, the second method, compiler forking, may show to be the most exciting future tool.

Chapter 7

Related Work

Rust's promise of memory safety and concurrency guarantees has attracted a lot of interest on this adaptation of C programs into Rust. Several approaches have been proposed to automate or assist this translation, each addressing the inherent challenges posed by C's lack of safety guarantees and Rust's strict ownership model.

Zhang et al. propose a static analysis framework that retypes C pointers into safe Rust abstractions like Box and &mut, enabling large-scale migration with reduced use of unsafe code [18]. Emre et al. propose a monotonic ownership model to reduce aliasing as a basic barrier to safe translation, showing that accurate ownership inference often fails in the presence of pointer aliasing [19]. By using pattern based pointer analysis [20], Hardekopf et al. further streamline the translation pipeline by converting dangerous Rust into safe Rust. While these approaches focus on generating safe Rust source, other work shifts toward IR-level modeling. Using annotated source transformations [21], Silva et al. modify Rust's MIR borrow-checking rules to C; Michael Ling et al. create a rule based transpiler to progressively rewrite dangerous C code into safer Rust [22].. Li et al. complement these approaches by doing a user study emphasizing the pragmatic challenges developers encounter when converting C to Rust, particularly in regard to ownership and lifetimes [23].

Unlike these tools and studies, our work focuses on directly creating a MIR-style intermediate representation from C code using Clang AST Traversal, so enabling fine-grained tracking of ownership semantics, control flow, and variable lifetimes, without depending on Rust compilation.

Chapter 8

Conclusion

This thesis investigated the creation of a stationary analysis and translation tool based on Mid-level Intermediate Representation (MIR) modeled after Rust compiler from C source code. Given the limitations of C programming language in terms of memory safety, lifetime tracking, and overflow detection, we investigated how such constructs could be encoded in a lower-level, analyzable from inspired by Rust’s internal compilation stages.

We developed and tested a prototype tool in C++ using Clang's Abstract Syntactic Tree (AST) and LibTooling frameworks in order to solve this challenge. The tool performs structural traversal of C code and emits output in a MIR-style format that includes control flow via basic blocks, variable tracking through scoped registers, arithmetic with overflow checks, and support for Rust-style formatted printing. Using custom logic and deferred code generation techniques reflecting Rust's internal behavior, key C constructs including functions, arithmetic expressions, while loops, and conditionals were translated into MIR equivalents.

We show throughout this work that it is possible to replicate semantic structure of C programs in a Rust-like MIR form, so facilitating deeper investigation and providing the basis for safe refactoring and possible cross-language translation. Using a set of representative C programs, our evaluation revealed that the produced output in both structure and logic quite closely matches real Rust MIR.

The tool is not feature-complete, but the results show its efficiency in managing fundamental language components and show future expansion possibility. Its usability might be raised even more by extensions including support for structs, pointers, for-loops, and switch statements as well as better optimizations. Ultimately, this work lays

the groundwork for hybrid tools that bring Rust's safety model to legacy C code via compiler-level translation and analysis.

References

[1] Kinza Yasar, “What is C (programming language)?”, TechTarget. https://www.techtarget.com/searchwindowsserver/definition/C

[2] Fernando Diaz, “How to secure memory-safe vs. manually managed languages”, GitLab, Mar. 14, 2023. https://about.gitlab.com/blog/2023/03/14/memory-safe-vs-unsafe/

[3] EC-Council University, “5 Most Popular Programming Languages for Cybersecurity”, eccu. https://www.eccu.edu/blog/most-popular-programming-languages-for-cybersecurity/

[4] Steve Klabnik, Carol Nichols, and Chris Krycho, with contributions from the Rust Community, “The Rust Programming Language”, rust-lang.org, Feb. 17, 2025. https://doc.rust-lang.org/book/title-page.html

[5] https://www.rust-lang.org/

[6] JSDevJournal, “Rust's Ownership System: 10 Things You Should Know”, Hackernoon, Sep. 4, 2023. https://hackernoon.com/rusts-ownership-system-10-things-you-should-know

[7] Kornel., “Speed of Rust vs C”, https://kornel.ski/rust-c-speed

[8] Syed Murtza, “Zero-Cost Abstractions in Rust: Power Without the Price”, Dockyard, Apr. 15, 2025. https://dockyard.com/blog/2025/04/15/zero-cost-abstractions-in-rust-power-without-the-price

[9] Sara Verdi, “Why Rust is the most admired language among developers”, GitHub, Aug. 30, 2023. https://github.blog/developer-skills/programming-languages-and-frameworks/why-rust-is-the-most-admired-language-among-developers/

[10] Mohammed Tawfik, “Rust Language: Pros, Cons, and Learning Guide”, Medium, Feb. 19, 2024. https://medium.com/@apicraft/rust-language-pros-cons-and-learning-guide-594e8c9e2b7c

[11] c2rust. https://github.com/immunant/c2rust

[12] Niko Matsakis, “Introducing MIR”, RustBlog, Apr. 19, 2016. https://blog.rust-lang.org/2016/04/19/MIR/

[13] Bala Priya C, “Abstract Syntax Tree (AST) - Explained in Plain English”, Dev.to, Dec. 28, 2022. https://dev.to/balapriya/abstract-syntax-tree-ast-explained-in-plain-english-1h38

[14] Gunashree RS, “ASTs Meaning: A Complete Programming Guide”, Devzery, Aug. 22, 2024. https://www.devzery.com/post/asts-meaning

[15] Shrey Vijayvargiya, “What is an Abstract Syntax Tree in Programming?”, Medium, Apr. 7, 2024. https://javascript.plainenglish.io/what-is-an-abstract-syntax-tree-in-programming-2f5d8a8d6f72

[16] “Introduction to the Clang AST”, Clang. https://clang.llvm.org/docs/IntroductionToTheClangAST.html

[17] “LibTooling”, Clang. https://clang.llvm.org/docs/LibTooling.html

[18] Hanliang Zhang, Christina David, Yijun Yu, Meng Wang, “Ownership Guided C to Rust Translation”, ResearchGate, Jul. 2023. https://www.researchgate.net/publication/372409182_Ownership_Guided_C_to_Rust_Translation

[19] Mehmet Emre, Peter Boyland, Aesha Parekh, Ryan Schroeder, Kyle Dewey, Ben Hardekopf, “Aliasing Limits on Translating C to Safer Rust”, ResearchGate, Apr. 2023. https://www.researchgate.net/publication/369863341_Aliasing_Limits_on_Translating_C_to_Safe_Rust

[20] Mehmet Emre, Ryan Schroeder, Kyle Dewey, Ben Hardekopf, “Translating C to Safer Rust”, ResearchGate, Oct. 2021. https://www.researchgate.net/publication/355438064_Translating_C_to_safer_Rust

[21] Tiago Silva, Joao Bispo, Tiago Carvalho, “Foundations for a Rust-Like Borrow Checker for C”, ResearchGate, Jun. 2024. https://www.researchgate.net/publication/381588695_Foundations_for_a_Rust-Like_Borrow_Checker_for_C

[22] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R. Cordy, Ahmed E. Hassan, “In Rust We Trust – A Transpiler from Unsafe C to Safer Rust”, ResearchGate, Oct. 2022.

https://www.researchgate.net/publication/364416328_In_rust_we_trust_a_transpiler_from_unsafe_C_to_safer_rust

[23] Ruishi Li, Bo Wang, Tianyu Li, Prateek Saxena, “Translating C to Rust: Lessons from a User Study”, ResearchGate, Jan. 2025 https://www.researchgate.net/publication/390111014_Translating_C_To_Rust_Lessons_from_a_User_Study

[24] MSRC Team, “A proactive approach to more secure code”, Microsoft, Jul. 16, 2019. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

[25] “Memory safety”, The Chromium Projects. https://www.chromium.org/Home/chromium-security/memory-safety/