UPPSALA
UNIVERSITET

IT kDV 26 014
Degree project 15 credits
June 2026

Extracting Representative Rust Benchmarks from GitHub Code

Misha Magnusson

Bachelor's Programme in Computer Science

UPPSALA
UNIVERSITET

Extracting Representative Rust Benchmarks from GitHub Code

Misha Magnusson

Abstract

Software written with users in mind should be tested on benchmarks representative of what those users input. Existing benchmark suites are not necessarily representative of real-world code. In this thesis I present a program that, given a function in a Rust codebase, resolves all its dependencies and extracts it into a self-contained project that can be built with standard Rust tooling. I evaluate my program on both a synthetic project, and empirically on a dataset of representative functions mined from GitHub, demonstrating how to generate representative benchmark suites for Rust. The results from the empirical evaluation show that while my approach is successful on non-trivial projects, extracting functions from code "found in the wild" requires implementing support for a larger set of Rust language features than my implementation currently support.

Faculty of Science and Technology

Uppsala University, Uppsala

Supervisor: Noé De Santo Subject reader: Andrea Gilot

Examiner: Johannes Borgström

Contents

1 Introduction **4**

2 Background **5**

3 Program design **8**
3.1 Overview . . . 8
3.2 Extraction pipeline . . . 8

4 Evaluation **14**
4.1 Synthetic project testing . . . 14
4.2 Empirical evaluation . . . 15
RQ1 To what extent is this approach effective in producing compilable benchmarks? . . . 17
RQ2 Which language features require manual intervention after extraction? 18
RQ3 What are the structural characteristics of the benchmarks extracted using this approach? . . . 19

5 Related Work **21**

6 Discussion and Conclusions **22**

Chapter 1

Introduction

Benchmark suites are designed to reproducibly, systematically, and robustly test and evaluate software. Yet many existing benchmarks are synthetic, hand-picked, or overly simplified. Such programs often fail to capture the complexity and structural patterns found in real-world code. Therefore, creating benchmarks representative of real-world code is crucial to ensure software is tested on input similar to what it will encounter after testing. Benchmarking Rust is today more than ever useful as more software is being written in Rust as well as being incorporated in already existing software, notably the Linux kernel. Because of its "safe" nature compared to C huge efforts are made to port C code to Rust [1, 2].

There already exist benchmark suites focused on representativeness [3, 4]. A shortcoming of some of the existing suites are that they use GitHub stars as a popularity metric to chose the programs for their suites [5, 3]. However, as explained by Maj et al. [6], projects with many stars are not representative of the general population.

In this thesis, I build a pipeline for extracting self-contained Rust benchmarks from GitHub repositories (Chapter 3). Given a function from a repository containing Rust code, my implementation extracts all its dependencies (both auxiliary functions and types) and synthesizes relevant imports from external libraries. The extracted code is packaged in a stand-alone project that users can build with Cargo[1]. Using a representative dataset of Rust functions, *e.g.*, extracted with tools like Scyros [7], my approach enables the generation of representative self-contained Rust benchmarks.

In my evaluation (Chapter 4), I show that my approach works on synthetic but complex examples. I also empirically test it on 385 functions drawn from a representative dataset of GitHub repositories written in Rust and discuss the remaining challenges to address to enable extraction on code found "in the wild" (Chapter 6).

On LLM usage. During this thesis, I used Large Language Models (LLMs) as a coding aid, *e.g.*, for understanding lifetime and ownership notation as well as general syntax. I also occasionally used LLMs for polishing the writing. Overall, LLMs served only as a "copilot" throughout this thesis, never making any executive decisions on my behalf.

[1] https://doc.rust-lang.org/cargo/

Chapter 2

Background

This chapter introduces Rust, as well as technical concepts such as Abstract Syntax Trees and the Language Server Protocol, which I regularly refer to in the chapter on program design (Chapter 3).

Rust. Rust is a programming language designed with a strong emphasis on memory safety and performance. I chose Rust for this thesis because it provides a standardized toolchain, allowing me to focus my efforts on coding the extractor itself. Compared to extracting benchmarks in C [7], the use of Rust eliminated the need for custom build scripts or manual dependency handling.

Cargo is Rust's build system and package manager, and makes compilation uniform across mined projects, whereas C lacks this standardization. This uniformity is critical when mining diverse projects, as it ensures consistent behaviour regardless of how each project is configured. In Cargo projects, the `lib.rs` file serves as the project's root, used to declare modules within the project, making them visible to Cargo.

External and internal libraries (crates) are imported into Rust projects via `use` items, referred to in this thesis as *use statements*. Additionally, the specific type for which a method is defined within an `impl` block is called the *impl context*. Figure 2.1 illustrates both importing external and internal dependencies with use statements, as well as the method `new` defined in an `impl` block.

Finally, the Rust compiler uses the LLVM compiler infrastructure, which means the extracted benchmarks can be used as both Rust and LLVM inputs for a wide range of tasks such as differential testing or compiler optimization.

Abstract Syntax Tree. An abstract syntax tree (AST) is a tree-like datastructure that represents the structure of a program. Each node makes up a component of the program, such as variable declarations, return statements, if-statements, etc. The relationship between the nodes denotes the relationship between the components in the program.

For example, the expression `x == 5 + 3` would be represented as an AST with an equality comparison node at the root. As shown in Figure 2.2, its left child is a node containing the identifier `x`, and its right child is a node representing the addition operation, which itself has two child nodes for the numeric literals 5 and 3.

test.rs

```rust
use crate::test_utils::Point;
use regex::Regex;

fn my_point() -> Point {
    let hello = Regex::new("hi").unwrap();
    return Point::new(0.0, 0.0);
}
```

test_utils.rs

```rust
struct Point {
    x: f64,
    y: f64,
}

impl Point {
    fn new(x: f64, y: f64) -> Self {
        Point { x, y }
    }
}
```

Figure 2.1: Example program demonstrating Rust use statements and impl context. The `use` statement imports the `Regex` type, and the `impl Point` block defines the `new` method for the `Point` type.

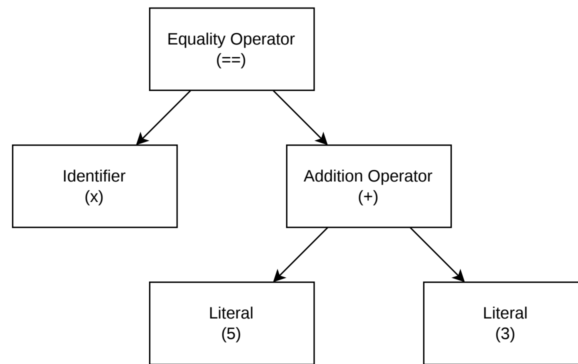

Figure 2.2: Abstract Syntax Tree (AST) representation for the expression `x == 5 + 3`.

JSON-RPC. JSON-RPC stands for JavaScript Object Notation - Remote Procedure Call. The intention of this protocol is to allow remote procedure calls, for example calling a method on a remote server. The data being transfered is in JSON format, hence its name, JSON format meaning the data is structured in key/value pairs, also supporting the use of ordered lists. Using this protocol the user can pass parameters to the method being called, and even receive results of the invoked method.

Language Server Protocol. The Language Server Protocol (LSP) is a communication protocol between a programmer's code editor and a server providing support in development to the user. It is built upon JSON-RPC for communication between the editor and the language server. A key feature of LSP is that it automatically compiles code in the background as the user edits, continuously analysing the program without requiring manual compilation. This allows the language server to provide real-time feedback such as code completion, syntax highlighting, and compiler warnings or errors. In this thesis I use Rust's official language server `rust-analyzer`[1] to extract the target function thanks to the information provided by the LSP.

To facilitate communication with the language server, I used the Rust library LSP-bridge[2] to interact directly with `rust-analyzer`. This library provides a high-level abstraction with typed, native Rust functions for common LSP utilities. An example is the `get_document_symbols` function, which requests a complete list of all document symbols

[1]https://rust-analyzer.github.io/
[2]https://docs.rs/lsp-bridge/latest/lsp_bridge/

within a source file. In the Language Server Protocol, a document symbol represents a distinct named element in the source code. These symbols include components such as functions, methods, modules, implementations (`impl` blocks), types, and fields.

Chapter 3

Program design

This section describes how my program extracts a target function into a self contained output repository containing the target function along with dependencies to preserve its behaviour. The goal is to produce a repository that compiles and that contains the target function isolated, keeping only the code necessary to preserve the behaviour of the target function.

3.1 Overview

The pipeline starts by extracting the target function's source code. Next, it recursively identifies all function dependencies using the LSP, extracting each one. Simultaneously, my program identifies all custom types used in the function and its dependencies, extracting their definitions. My program also tracks which implementation blocks these functions belong to. Finally, my program initializes a new Cargo project, writes the extracted code with proper module structure and imports, and produces a self-contained repository that compiles and contains only the code necessary to preserve the behaviour of the original function.

To extract a function using my program the user specifies the function's position (line and column number), the path to the file it is in and the project's root path. See box 1 in Figure 3.1.

3.2 Extraction pipeline

LSP-bridge Initialization. My program utilizes LSP-bridge to communicate with `rust-analyzer`. I chose LSP-bridge for its high-level abstraction. Specifically, it provides typed Rust functions for common LSP operations such as retrieving document symbols and locating function definitions, eliminating the need to manually construct and parse JSON-RPC messages. The code for the initialization process is shown in Figure 3.2.

At this point of the program: A live LSP connection is established and ready to query the project. LSP-bridge is connected to `rust-analyzer`, which has initialized the target project's root path.

Function Extraction. (3B) Using LSP-bridge's `get_document_symbols` my program

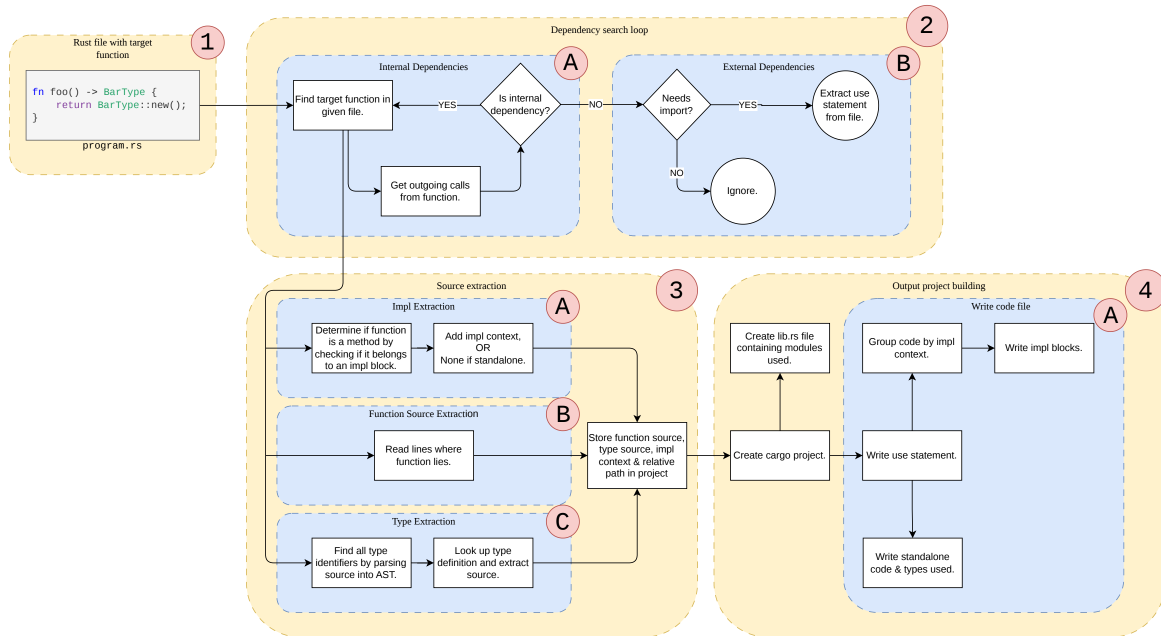

Figure 3.1: High-level program design.

main.rs

```rust
// Create a new bridge
let mut bridge = LspBridge::new();
let uri = format!("file://{}", filename);

// Configure rust-analyzer server
let config = LspServerConfig::new()
    .command("rust-analyzer")
    .root_path(PathBuf::from(root_path));

// Register and start the server
let server_id = bridge.register_server("rust-analyzer", config).await?;
bridge.start_server(&server_id).await?;
```

Figure 3.2: Rust code that initializes a connection to the rust-analyzer LSP server.

extracts all of the symbols in the file containing the target function. Each symbol has associated metadata, my program utilizes three fields:

- `name`: The symbol's identifier (e.g., function or type name).
- `range`: The full span enclosing the entire definition and body of the symbol. For a function, this covers everything from the attributes and visibility modifiers to the closing curly brace (e.g., `pub fn foo() {...}`). This span is used to extract the symbol's complete source code.
- `selection_range`: The region of text highlighted when navigating to this symbol in an editor (e.g., just the function name, excluding the body), it is contained within `range`.

My program finds the symbol whose selection_range contains the target position (line and column), then retrieves its full range. Using the retrieved range, my program reads the corresponding lines from the document to extract the function's source code. An example

of extracting the function `test_data` can be seen in Figure 3.3.

An alternative approach would be identifying the target function by its name. However, my program uses function position because Scyros, the tool that my program extends, provides function line and column. Another concern with that approach is that multiple functions can share the same name within a document. For example, the same method `new` can appear in different `impl` blocks, and the function name alone cannot distinguish them.

Another approach would be to use fully qualified function names that contain the crate, module and type context, such as `crate::module::Struct::function_name`. However, because Scyros operates strictly on a per-file basis, it only provides positional coordinates rather than qualified paths. Consequently, resolving qualified names is unnecessary, as Scyros does not provide them in the first place.

At this point of the program: The target function's source code is extracted along with its line range in the document. The program now proceeds to recursively identify and extract the function's dependencies.

Dependency Extraction. (2) I use LSP-bridge's `get_outgoing_calls` function to get all of the outgoing function calls made by the target function. The function extraction step is applied recursively to resolve and extract all dependencies. Before the recursive step is applied the called function's filepath is checked to see if it exists within the project root. If the filepath exists within the project root, then it is an internal dependency (2A) and the extraction loop is recursively called using that filepath.

Else, if the filepath exists outside the project root, then it is an external dependency (2B). Next, my program checks whether it requires a use statement (standard libraries do not need use statements), if not, the dependency is ignored. In the case that the dependency requires a use statement, the crate name is extracted from the filepath. The crate name is then used to pattern match corresponding use statements in the file, and extracted (e.g., for the crate `regex`, all lines containing `use regex::` are extracted). Note that Cargo downloads and installs external crates automatically upon building the project.

See Figure 3.3 for an example of extracting the internal dependencies `new`, `add_string`, and `add_number`.

At this point of the program: All internal function dependencies are recursively extracted, and external dependencies are identified and catalogued. The program now has the full call hierarchy with both internal and external function dependencies resolved.

Type Extraction. (3C) For each function source that is extracted my program parses it into an AST using the `parse_str` function from the syn[1] crate. I decided to use the syn crate for AST extraction because it is well documented, and is also widely adopted with 3300 GitHub stars as of May 2026. My program traverses the AST to find types used in the function signature, body and return value. Then it collects type identifiers from the AST, queries LSP-bridge to locate their defining documents, and extracts their source by iterating over the document symbols to find the matching type identifier. Specifically, my program only extracts custom user-defined types such as structs, enums and unions.

[1]https://docs.rs/syn/latest/syn/

Primitive types from the standard library (e.g., `usize`, `String`, `Vec`) are not extracted because they are implicitly available in any Rust project. An example demonstrating extracting the type `TestFixture` is shown in Figure 3.3.

At this point of the program: All types used by the extracted functions are identified, and their definitions are extracted from the source project. The program now has a complete set of functions and types needed to preserve behaviour.

Impl Extraction. (3A) For each function source that is extracted, my program stores its implementation context. The implementation context is the `impl` identifier it belongs to or 'None' if it is a stand-alone function. This implementation context determines whether each function is written as part of an `impl` block or as a stand-alone function in the output project.

See an example of extracting the `TestFixture impl` block in Figure 3.3.

At this point of the program: Each extracted function is classified as either a stand-alone function or a method belonging to a specific `impl` block. This context is used to properly group code when writing the output project.

Output Project Building. (4) Each extracted function, implementation and type is stored in a structure called `ProjectOutput`. The `ProjectOutput` structure contains a HashMap which maps the output file path to the `ExtractedCode` structure. Each `ExtractedCode` structure contains three fields:

- `code`: The source code of the extracted function or type definition
- `types_used`: A list of type identifiers used to determine which types the extracted code depends on. Only the identifier is stored rather than the entire type definition to make this structure lighter, the actual definitions are extracted during the write phase
- `impl_context`: Either the identifier of the implementation or 'None' if stand-alone function.

When extraction is complete, the `write_project_to_disk` function coordinates disk writing such that the output structure mirrors the source project's directory structure.

First, the `lib.rs` file is generated to serve as the root module file that Cargo uses to identify which modules exist in the output project. Without `lib.rs`, Cargo would not know which modules to compile. My program generates `lib.rs` by scanning all extracted file paths to identify unique top-level module directories and files, then writing `pub mod` declarations for each one (e.g. writing `pub mod test;` for the file `src/test.rs`).

An alternative is to track modules during the extraction phase and generate `lib.rs` from that data. I chose to generate module files during the write phase so that the extraction phase does not need to track which module each extracted item belongs to, thus simplifying the extraction phase of my program (3).

Note that my program does not track whether the target project's modules are public or private. While this does not affect the functionality of the generated code, the original developer's intent regarding module visibility is not preserved.

The final stage of my program involves writing the extracted code to disk (4A). For

each target file, the system collects all type identifiers used across every extracted code element. It then checks which types are defined in other modules. For types originating from other files within the target project, the system automatically generates use crate::<module>::<TypeName> statements, using the module path derived from the source filepath.

Use statements are generated during the write phase rather than during extraction because the extraction phase lacks the full context of which modules and crates are present in the final output project. By delaying use generation to write-time, all extracted modules and crates are already identified, making it straightforward to determine the correct imports. These use statements are prepended to the file before any code content.

My program divides the extracted code into two categories: stand-alone items, such as functions and type definitions, and methods belonging to impl blocks. Stand-alone code is written first (as shown for the type definition and function in Figure 3.3). For each type, my program first generates the implementation header: impl <TypeName> {, then each method associated with that implementation, and finally a closing curly bracket (}).

Once all individual code files are written, the system generates the necessary module declaration files by scanning all extracted file paths to identify the nested directory structure. For each directory and file not directly under the project root containing extracted code, a mod.rs file is created to declare all modules within that directory as public.

For example, if src/testing_code/test.rs and src/testing_code/test_utils.rs both contain extracted code, a src/testing_code/mod.rs file is generated containing pub mod test; and pub mod test_utils;.

An alternative approach would be to place all module declarations directly in lib.rs, allowing it to define the entire project's nested module structure with inline declarations such as pub mod testing_code { pub mod test; pub mod test_utils; }. This would eliminate the need for separate mod.rs files throughout the output directory. However, I chose the current approach because by generating mod.rs files for each directory after the code is written, I avoid needing to pre-compute the full nested module tree structure upfront. My current implementation simply writes files and folders without tracking their hierarchical relationships, then declares them to Cargo using simple, flat lib.rs and mod.rs files with no nested logic. This avoids the complexity of parsing the project hierarchy and generating properly-nested module declarations in lib.rs.

At this point of the program: The extracted code is written to disk within the proper module structure. The output project is now complete with lib.rs and mod.rs files, and all extracted functions and types are organized in their original directory structure.

source_project/test.rs

```
1 use super::test_utils::*
2
3 ...
4
5 fn test_data() -> TestFixture {
6   let mut fixture = TestFixture::new();
7   fixture.add_string("hi".to_string());
8   fixture.add_number(100);
9   return fixture;
10 }
11
12 ...
```

source_project/test_utils.rs

```
33 ...
34
35 pub struct TestFixture {
36   strings: Vec<String>,
37   numbers: Vec<i32>,
38   floats: Vec<f64>,
39 }
40
41 impl TestFixture {
42   fn new() -> Self { ... }
43   fn add_string(&mut self, s: String)
44     { self.strings.push(s); }
45   fn add_number(&mut self, n: i32)
46     { self.numbers.push(n); }
47   fn add_floats(&mut self, n: f64)
48     { self.floats.push(n); }
49   fn clear(&mut self) {
50     self.strings.clear();
51     self.numbers.clear();
52     self.floats.clear();
53   }
54 }
55
56 ...
```

(a) **Input:** source project containing function to extract

output_project/test.rs

```
1 use crate::test_utils::TestFixture;
2
3 fn test_data() -> TestFixture {
4   let mut fixture = TestFixture::new();
5   fixture.add_string("hi".to_string());
6   fixture.add_number(100);
7   return fixture;
8 }
```

output_project/test_utils.rs

```
1 pub struct TestFixture {
2   strings: Vec<String>,
3   numbers: Vec<i32>,
4   floats: Vec<f64>,
5 }
6
7 impl TestFixture {
8   fn new() -> Self { ... }
9   fn add_string(&mut self, s: String)
10     { ... }
11   fn add_number(&mut self, n: i32)
12     { ... }
13 }
```

(b) **Output:** output project containing extracted function along with its dependencies

Figure 3.3: Example showing extraction of the `test_data` function. When extracting the target function `test_data`, the tool extracts the user-defined type `TestFixture` and only the implementation methods that are actually used by the function. The left panel shows the original source files containing `test_data` which calls `TestFixture::new`, `add_string`, and `add_number`. The right panel shows the extracted output: the function, the type definition, and only the three methods that `test_data` depends on. Note that `add_floats` and `clear` are not extracted because they are not called by `test_data`.

Chapter 4

Evaluation

I first test my program on a complex synthetic project to verify whether it handles hand-picked Rust language features. I complement this with an empirical evaluation on real-world repositories from GitHub to assess how it performs on programs found "in the wild" and identify remaining practical challenges.

4.1 Synthetic project testing

I test my progam by extracting functions from an LLM-generated complex project. The functions in this project progress through increasing complexity, testing the extraction pipeline's ability to successively handle more sophisticated language features.

Synthetic Project Contents. The project consists of 1102 lines of code in total including all subdirectories and files. The directory structure of the testing project is shown in Figure 4.1. The functions that are used to test my extraction pipeline are contained in `test.rs`.

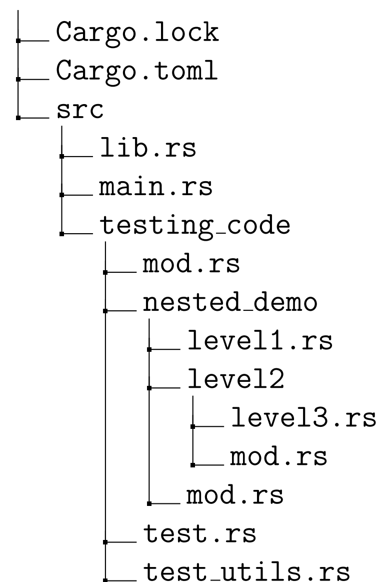

Figure 4.1: Directory structure of the synthetic testing project.

The first and simplest function takes no parameters and returns an integer. Subsequent functions increase in complexity by taking parameters such as strings and integers and returning them. Some of these functions also make use of standard macros (`format!`) for string manipulation.

Next added complexity is the use of mutable variables (`mut`), and functions with borrowing (`&`) and explicit lifetime notation(<`'a`>). After that I defined custom structs along with implementations for them with methods such as (`new`).

I introduced functions with generics types using the <T> notation. Then, I gave generic types trait bounds, such as `Display` and `Clone`. I also wrote

some custom trait definitions, to test that. I followed that with custom enum definitions with associated methods (`impl`).

Next, I wrote functions making use of functions from external crates were introduced (such as `Regex` and `rand`) by writing `use` statements. Then, I wrote some functions that return anonymous functions (closures).

To test extraction of functions with dependencies to different modules in the testing project, I created `test_utils.rs` containing various functions. Then I imported these function into `test.rs` with `use` statements. Next, I wrote functions that called these newly imported functions.

To test deeper nested dependencies I added the `nested_demo` folder containing another Rust file with some functions, that I imported into `test.rs` and tested. Finally, I created more complex functions making use of multiple of the previously mentioned language features simultaneously.

Evaluation Results. My program could successfully extract **13** out of the **76** functions defined in `test.rs`. The **63** functions the program could not extract use features my implementation does not support yet, such as polymorphism or references to methods that implements generic types. It also fails on methods where the type they implement is not directly referenced in the method body or signature. This last case is due to a bug where the extractor detects that a method belongs to an `impl` block, but since the type being implemented is not detected in the method, it is not extracted.

Figure 4.2 shows the result of the extraction of the function `test` from the synthetic testing project

4.2 Empirical evaluation

I also empirically evaluate my approach on projects I downloaded from GitHub and investigate the following research questions:

- **RQ1.** To what extent is this approach effective in producing compilable benchmarks?
- **RQ2.** Which language features require manual intervention after extraction?
- **RQ3.** What are the structural characteristics of the benchmarks extracted using this approach?

During each extraction my program collects the following statistics for each extracted benchmark:

- whether it compiles, or the number of build errors if it does not;
- list and count of extracted functions and methods;
- list and count of extracted user-defined types;
- list and count of external dependencies present in `Cargo.toml`;
- number of files traversed during extraction;

(a) **Input:** Source file with target function

(b) **Output:** Generated project

Figure 4.2: Full extraction example of target function `test`. The left panel shows the original source code file containing the target function. The right panel shows all extracted files.

- lines of code (LOC) in both the original project and extracted benchmark;
- LOC for each extracted function;
- whether the extracted benchmark contains unsafe Rust code.

The extraction success rate addresses RQ1, while build errors allow me to answer RQ2. I complement the latter with a manual inspection of failure causes for benchmarks where extraction did not succeed. Most of the collected statistics structurally characterise the extracted benchmarks, allowing me to answer RQ3.

Data Preprocessing Pipeline. I started with a dataset of approximately 1.2 million GitHub repositories that had already been preprocessed [8]. The preprocessing steps included collecting random repository IDs, removing forked repositories, filtering for repositories with at least 2 months of activity between creation and last push, removing repositories smaller than 50 KB, and collecting metadata such as language used.

Using Cochran's formula [9] for sample size estimation in large populations, I select 385 function to construct the evaluation dataset, ensuring statistical representativeness at a 95% confidence level with a 5% margin of error. To ensure these functions were well-distributed across different codebases rather than concentrated in a few projects, I downloaded 147 unique repositories containing Rust using the Scyros pipeline. This download process also generated a CSV file mapping each GitHub repository ID to its corresponding local directory path. Since my program extracts Rust code, I filtered out non-Rust functions from the sampled files.

Next, I deduplicated files with Scyros. This step isolates authentic, user-written code and avoids double counting by removing copy-pasted files, build artifacts, or automated files that might otherwise skew the statistics and compromise representativeness.

I then used Scyros to extract all function definitions from the downloaded repositories. This process mapped each function definition to its corresponding GitHub repository ID and precise location, including the file path, line, and column offset.

Finally, this leaves my program with the required identification for extracting a function: project path, file path, function line, and character offset.

Experimental Setup. I conducted all experiments on a laptop with the following specifications:

- CPU: 11th Gen Intel® Core™ i5-11300H *(4 physical cores, 8 logical threads)*
- RAM: 16,0 GiB
- OS: Ubuntu 24.04.4 LTS

The Rust version I used was **rustc 1.91.1** and Cargo version **1.91.1**

RQ1 To what extent is this approach effective in producing compilable benchmarks?

To evaluate how effectively my program successfully extracts code from real-world projects, Table 4.1 summarizes the results from benchmarks that compiled successfully. These

Table 4.1: Compilable Benchmarks Summary

Metric	**Value**
Benchmarks Count	7
Unsafe Rust Count	0
Avg. Unique User-Defined Types	0.43
Avg. Stand-alone Functions	0.57
Avg. Methods	0.43
Avg. Total Files Processed	1.14
Avg. Total External Dependencies	0.00
Avg. External Crates Count	0.00
Avg. Reduction Percentage	99.85%
Avg. Extraction Time (ms)	5,089.57
Avg. Total Benchmark Generation Time (ms)	8,597.29

benchmarks correspond to the functions that my program could successfully extract and, once extracted, could compile. To complement this, a summary of the extraction time and structure of the non-compilable benchmarks is shown in Table 4.2. These benchmarks correspond to the functions that my program could successfully extract but, once extracted, could *not* compile.

For the benchmarks that my program could not extract, a summary of the reasons is shown in Table 4.3. As seen in the table, 113 out of 114 (99.12%) of the benchmarks could not be extracted because there was no function on the position given by Scyros. Therefore, these failed extractions are caused by an error of Scyros. The one failed extraction was due to the following error: `Error("expected ‘fn‘")`. Upon inspection my program attempted to extract the `default` function. The extraction failed because the function is a predefined, standard `HashMap` constructor, not a function definition which Scyros is supposed to provide and what my extractor expects.

To summarize, 2.7% of the benchmarks that were produced by my program could be compiled, respectively 97.3% of the benchmarks could not compile. On average, the extraction time took 5.98 seconds to extract a benchmark and write it to disk, excluding things like LSP initialization and the extractor program to start up.

RQ2 Which language features require manual intervention after extraction?

From the functions mined from GitHub, the language features that caused my extraction pipeline to fail can be seen in Table 4.4, in total there were 63 failed benchmarks that were due to an unsupported language feature. The rest of the non-compiling benchmarks failed due to other causes.

The largest categories were the use of the Rust keyword `mod` as an identifier and unstable compiler-only constructs. The former category are due to a bug in my program when extracting code coming from a `mod.rs` file. My program first extracts function into a file called `mod.rs`, then writes a new `mod.rs` file to reference that module. Thus overwriting

Table 4.2: Non-Compilable Benchmarks Summary

Metric	Value
Benchmarks Count	253
Unsafe Rust Count	15
Avg. Unique User-Defined Types	0.51
Avg. Stand-alone Functions	0.53
Avg. Methods	0.47
Avg. Total Files Processed	1.43
Avg. Total External Dependencies	0.00
Avg. External Crates Count	0.00
Avg. Reduction Percentage	99.46%
Avg. Extraction Time (ms)	6,889.25
Avg. Total Benchmark Generation Time (ms)	12,033.99

Table 4.3: Summary of the benchmarks that could not be extracted

Metric	Value
Total Error Count	114
Percentage No Function Found	99.12%
Percentage Other	0.88%

the first file and resulting in a `mod` file containing just `pub mod mod;`, which is syntatically incorrect Rust code. This is a scenario which was not covered by my synthetic project in Section 4.1.

The unstable compiler-only construct cases were not ordinary application-level errors, but code that depended on nightly-gated or compiler-internal Rust features, such as the `allocator_api` attribute and the `Allocator` trait, or other unstable internal items used in low-level/generated code. Smaller groups involved missing procedural macros or derive expansions, where the code relied on annotations such as `#[derive(Serialize, Deserialize)]` and `#[serde(...)]` but the corresponding macro definitions or crate dependencies were unavailable. One benchmark failed specifically because it used the nightly-only `extern "x86-interrupt"` ABI (Application Binary Interface), which is not supported on stable Rust.

An important note is that it is not guaranteed that the benchmark would become compilable upon fixing support for this langauge feature or fix the direct cause for the build errors, since there more errors can come after fixing the current ones.

RQ3 What are the structural characteristics of the benchmarks extracted using this approach?

The extracted benchmarks are structurally minimalist, typically spanning only a single file (∼1.28 files on average). They contain fewer than one unique user-defined type, stand-alone function, or method per benchmark. During the empirical study no external crates

Table 4.4: Summary of language-feature-related causes of benchmark compilation failure

Language Feature	**Count**	**Percentage**
Reserved keyword `mod` used as identifier	27	42.86%
Unstable compiler-only features	27	42.86%
Missing procedural macro or derive expansion	8	12.70%
Nightly-only experimental ABI	1	1.59%

were able to be extracted. Code reduction exceeded 99.4%, stripping away almost the entire original project. As Table 4.1 and Table 4.2 show, 15 out of the 260 extracted benchmarks (5.76%) made use of unsafe Rust.

Chapter 5

Related Work

Extraction. The extraction tool REM [10] implements the “Extract Method”, an algorithm that targets a snippet of Rust code and refactors it into a function within the same file. This approach does not extract target functions into stand-alone programs. Regardless the life-time and ownership repair algorithms introduced in the paper are of interest to this thesis. These repair algorithms deal with preserving the correctness of lifetime and ownership annotations which also is a goal in the extracted function of this thesis. Since REM refactors code within the same code document, it does not need to deal with dependencies, unlike this thesis.

Benchmarking. AnghaBench [5] is a suite containing one million real-world compilable C benchmarks extracted from Github. Its selection method favors large repositories with many stars which does not necessarily produce representative benchmarks. As argued by Maj et al. [6], popularity metrics such as GitHub stars are not strongly correlated with code quality or diversity. Another limitation of the AnghaBench approach is that, when extracted functions have missing dependencies, the framework may complete the program using stubbed code. This introduces artificial code that was not written by the original developers, which reduces the representativeness of the benchmark. This thesis aims to generate representative benchmarks by avoiding introducing artificial code and instead extract everything neccessary from the target code base.

Exebench [3] is a suite of benchmarks that are not only compilable, like in this thesis but also runnable, meaning the programs can be executed. The downside to their appoach is that unresolved dependencies like for example `#include "lib.h"` will be borrowed from other projects when not found in the original project. Notably, dependencies coming from repositories with more stars are prioritized. This constraint excludes a big portion of real-world code on GitHub, which can reduce representativeness of the suite. In this thesis benchmarks are more representative of real-world code with the trade-off of containing non-runnable programs.

Testing. The randomised Rust program generator RustSmith [11] generates entirely synthetic code suitable for differential testing. On the contrary, the test suites presented in this thesis focus on real-world code and representativeness. The RustSmith approach benefits testing of compilers while our approach is designed to do testing under realistic conditions.

Chapter 6

Discussion and Conclusions

Prior to this thesis I had no experience in Rust. Because of that, I spent a large portion of the first week learning Rust and looking at blog posts of how Rust compares to the programming languages I already knew.

Functions were randomly sampled from all functions found in the downloaded repositories. Larger repositories are overrepresented in this sample because they contribute more functions to the pool.

LSP-bridge's document symbols `range` field includes both line and column positions, which would allow for column-level precision when my program extracts source code. However, the current extraction logic operates only at the line level, reading entire lines spanning from the symbol's start to end. If multiple functions, types, or methods are defined on the same lines, this approach will extract unintended code alongside the target symbol. For example, if a function definition ends on the same line where another function begins, both would be extracted together. When my extraction pipeline encounters such code, the extraction pipeline crashes due to an "`unexpected token`" error and no output project is generated. This is a limitation of my current implementation, though it can easily be addressed by implementing column-aware extraction logic.

Conclusions. From the empirical evaluation of my program resulting in a 2.7% success rate of extracting a target function I conclude the problem I am solving is non-trivial.

Looking at the results from Section RQ2 the language features that caused benchmarks to *not* compile were those I did not yet implement support for. On the contrary, the language features that I implemented support for could successfully be extracted for the most part (excluding external dependencies). This leads me to believe that my approach works, but to increase the amount of successful extractions I must implement support for more language features. I believe implementing support for more language features can be done using the same method as for the ones already supported.

Answering Section RQ3 I noted that only 5.76% of the 260 extracted benchmarks made use of unsafe Rust notation. This suggests that the majority of code found in the wild does *not* use unsafe Rust notation and is therefore memory safe and data race free!

As discussed answering Section RQ1, the extraction time of benchmarks took 5.98 seconds, since this is a process that needs to communicate with an LSP, I consider that

effective.

For this approach to work, I would need to implement support for more language features, as well as fixing bugs in my current implementation.

Bibliography

[1] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, “Translating C to safer Rust,” *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3485498

[2] M. Ling, Y. Yu, H. Wu, Y. Wang, J. R. Cordy, and A. E. Hassan, “In Rust we trust: a transpiler from unsafe C to safer Rust,” in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE ’22. Association for Computing Machinery, pp. 354–355. [Online]. Available: https://dl.acm.org/doi/10.1145/3510454.3528640

[3] J. Armengol-Estapé, J. Woodruff, A. Brauckmann, J. W. d. S. Magalhães, and M. F. P. O’Boyle, “ExeBench: an ML-scale dataset of executable C functions,” in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, ser. MAPS 2022. Association for Computing Machinery, pp. 50–59. [Online]. Available: https://dl.acm.org/doi/10.1145/3520312.3534867

[4] L. N. Q. Do, M. Eichberg, and E. Bodden, “Toward an automated benchmark management system,” in *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2016. Association for Computing Machinery, pp. 13–17. [Online]. Available: https://dl.acm.org/doi/10.1145/2931021.2931023

[5] A. F. da Silva, B. C. Kind, J. W. de Souza Magalhães, J. N. Rocha, B. C. Ferreira Guimarães, and F. M. Quinão Pereira, “ANGHABENCH: A suite with one million compilable C benchmarks for code-size reduction,” in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. [Online]. Available: https://ieeexplore.ieee.org/document/9370322/

[6] P. Maj, S. Muroya, K. Siek, L. Di Grazia, and J. Vitek, “The Fault in Our Stars: Designing Reproducible Large-scale Code Analysis Experiments,” in *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), J. Aldrich and G. Salvaneschi, Eds., vol. 313. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 27:1–27:23. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.27

[7] A. Gilot, T. Wrigstad, and E. Darulova, “Floating-point usage on GitHub: A large-scale study of statically typed languages,” *Proc. ACM Program. Lang.*, vol. 10, no. OOPSLA1, Apr. 2026. [Online]. Available: https://doi.org/10.1145/3798203

[8] ——, "Floating-point usage on github: a large-scale study of statically typed languages (dataset)," Mar. 2026. [Online]. Available: https://doi.org/10.5281/zenodo.19242742

[9] W. G. Cochran, *Sampling Techniques*, 3rd ed. John Wiley & Sons, 1977.

[10] S. Thy, A. Costea, K. Gopinathan, and I. Sergey, "Adventure of a lifetime: Extract method refactoring for rust," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, Oct. 2023. [Online]. Available: https://doi.org/10.1145/3622821

[11] M. Sharma, P. Yu, and A. F. Donaldson, "RustSmith: Random differential compiler testing for Rust," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. Association for Computing Machinery, pp. 1483–1486. [Online]. Available: https://dl.acm.org/doi/10.1145/3597926.3604919