Static Analysis of Rust Error Propagation

THOMAS KAS, University of Twente, The Netherlands

The Rust programming language deals with errors in a different way than many popular languages. For example, languages such as Java have an exception system which interrupts the regular execution flow within programs. In this system, errors thrown by called functions are propagated by default, unless explicitly caught and handled.

In contrast, Rust wraps errors using its robust type system. This simplifies the error system as it does not interrupt regular program execution, which might make it easier to reason about. Furthermore, it requires developers to explicitly handle errors that might come up. The type used to wrap errors is a regular enumeration, **Result**, which has two possible values: **Result::Ok**; and **Result::Err**. The former wraps the resulting value in case of successful execution, whereas the latter wraps the error value otherwise. Developers can then interact with this type using Rust's many features such as pattern matching, or explicitly propagate error values using the try operator, '**?**'.

Moreover, Rust has a way to signal unrecoverable errors with the panic system. When used, this system unwinds the stack and halts execution. Because this system does break execution flow, it should be used sparingly by developers. It can be invoked both directly using the macro **panic!**, or by built-in functions such as **Result::unwrap**. This specific function either returns the value wrapped in **Result::Ok** or panics if the value was of type **Result::Err**.

We feel this approach to errors might lead to better error-handling practices among developers. However, no research has been done on how developers use these error systems. In this paper, we present a newly developed static analysis tool that can show the error propagation within Rust programs. Furthermore, we use our tool to conduct a small case study to explore how the error system is used in two open-source Rust programs. Here, we show that our tool is helpful in quickly identifying where and how the error system is used, as well as potential issues in its usage. We hope this tool enables further research into the ways the error system of the Rust language is used in practice.

Additional Key Words and Phrases: Rust, Static Analysis, Error Propagation

1 INTRODUCTION

Over the past 8 years, Rust has steadily ranked as the most loved/admired programming language on Stack Overflow's Developer Surveys [8, 9]. Its focus on safety, performance, and concurrency makes it ideal for use in critical systems. Comparing its performance to that of C, Zhang et al [11] found in their 2022 study that Rust is on average 1.77x slower. They found that this performance discrepancy was mainly caused by various runtime checks Rust implements. These runtime checks protect against problems such as accessing arrays out of bounds, integer overflow, and division by zero.

Along with such features, Rust also uses a relatively simple error system, which deviates from many popular languages. Rather than using a throw-catch system, it instead uses its robust type system to wrap errors. A function whose execution might be unsuccessful returns the `Result` enumeration. This is a regular type which follows all regular execution rules. This error system, or rather the lack thereof, simplifies execution flow.

41st Twente Student Conference on IT, July 5, 2024, Enschede, The Netherlands

We feel this approach to errors might lead to better error-handling practices among developers. However, no research has been done on how developers use these error systems. As such, we pose the following research question:

In what ways are errors created, handled, and propagated in the Rust programming language?

To help answer this question, we have created a static analysis tool to extract data on the propagation of errors within Rust programs. This tool creates a graph of the propagation of errors between functions within a program, and extracts specific metrics from there. The error propagation graph helps developers and researchers in quickly identifying where and how the error system is used. Furthermore, the metrics are useful for identifying some issues in the error handling strategy, such as errors that get propagated too many times before being handled.

In this paper, we present our tool, as well as our findings from using said tool on two open-source Rust programs. We first give background information on how Rust works in general, how it and other languages deal with errors, as well as how Rust is compiled in section 2. Next, we go over related work such as static analysis tools made for other programming languages in section 3. We then cover the definitions we used when creating our tool as well as how the tool is used in section 4. Next, we detail the implementation of our tool in section 5. We go on to use our tool for a case study, investigating two open-source Rust programs in section 6. Finally, we go over our conclusions in section 7, as well as future work suggestions in section 8.

2 BACKGROUND

In this section, we explore some basic Rust concepts, how exceptions work in most languages, as well as how Rust's errors work. Finally, we give an overview of the Rust compiler's steps.

2.1 Rust

Rust is a statically-typed, compiled language leading to high performance compared to interpreted languages. Furthermore, it is a memory-safe language without a garbage collector, which leads to low overhead compared to languages with a garbage collector. In this subsection we explore the basics of its type-inference and ownership model.

2.1.1 Type Inference. Since Rust is statically-typed, the types of variables are always known at compile-time [7]. Usually, variable types can be inferred based on their value and how they are used. In some cases however, explicit type information has to be given for the compiler to know what type a variable has. Type inference makes it easier to work with the language as a developer, however it complicates the compiler and static analysis where types are important. The following snippet shows two variables being instantiated. The type of the first variable can be inferred from its value, while the other needs to be explicitly given as the compiler needs to know what to parse the value into:

```
let value = "42";
let parsed: u32 = value.parse().unwrap();
```

2.1.2 Ownership. Rust achieves memory-safety without garbage collection through its unique ownership system [7]. The ownership system takes some getting used to, but its benefits outweigh its drawbacks. Essentially, each value within a Rust program has a single owner, the variable it belongs to. When this owner goes out of scope, the value is dropped. Ownership can be transferred, after which the original owner can no longer access the value. The following example shows the general concept:

```
fn main() {
    let s = String::from("hello");
// We own s
    use_string_ref(&s);
// We still own s as we only passed a reference
    use_string(s);
// We can not use s as we transferred ownership
}

fn use_string_ref(s: &str) {
// We only have a reference of s
}

fn use_string(s: String) {
// We now own s
}
```

2.2 Exceptions

In many languages exception handling follows concepts as proposed in a 1975 paper by Goodenough [6]. Here an *exception* refers to an abnormal state in execution. Whenever such a state occurs, an exception is *raised*. An *exception handler* can be defined to handle specific exceptions. An exception is *handled* when an exception handler finishes execution.

For example, in Java a method which might run into an exception signifies this in its signature as follows:

```
void exception() throws Exception
```

When using this method from another method, the call has to either be enclosed in a try block:

```
void catch_exception() {
    try {
        exception();
    } catch (Exception e) {
        // Handle e
    }
}
```

Or the method should also throw the same exception, in which case the exception is implicitly propagated:

```
void prop_exception() throws Exception {
    exception();
}
```

This automatic exception propagation might lead to cases where developers forget to handle certain exceptions where they meant to, erroneously propagating them.

So how does Rust deal with errors?

2.3 Rust's Errors

Rust does not follow these concepts. Instead of exceptions, it works with `Results` for recoverable errors. The `Result` type is a regular enumeration type, with 2 generic type arguments and 2 possible values. The first generic argument refers to the success type, values of which are wrapped in the first possible value `Result::Ok`. The second generic argument refers to the error type, values of which are wrapped in the second possible value `Result::Err`.

For example, consider the case where on successful execution a function would return a value of type `i32`, but on erroneous execution it would return an error simply called `Error`. Then the return type would be `Result<i32, Error>`. When returning the value upon successful execution, we need to wrap the integer in `Result::Ok`. Likewise, when returning an error value, we need to wrap the error in `Result::Err`. The complete function would look as follows:

```
fn result(successful: bool) -> Result<i32, Error> {
    if successful {
        return Ok(1);
    } else {
        return Err(Error);
    }
}
```

When we want to access the result of the function, we need to explicitly check whether execution was successful or an error was returned. This can be done using Rust's pattern matching features, for example:

```
let res: Result<i32, Error> = result(false);
match res {
    Ok(val) => {
        // Do something with val
    }
    Err(e) => {
        // Handle error
    }
}
```

Rust also provides an easy way to propagate such errors from a function, using the post-fix try operator: ?. This essentially desugars into a match statement, where Ok has its value extracted, and Err has the error value propagated. When the propagated error type does not match the return type of the function, Rust tries to convert the type using the `From` trait, if it is implemented. For example:

```
fn propagate() -> Result<(), OtherError> {
    match result(false) {
        Ok(val) => {
            let res: i32 = val;
            return Ok(());
        }
        Err(e) => {
            return Err(OtherError::from(e));
        }
    }
}
```

Can be implemented instead as:

```
fn propagate() -> Result<(), OtherError> {
    let res: i32 = result(false)?;
    return Ok(());
}
```

Because of the lack of error-handling code blocks, error handling happens among non-error-handling code. It can be dealt with in any way the developer wants. This further complicates static analysis of errors within Rust programs.

Since these errors are regular types, they do not break execution flow in any way. However, what if an unrecoverable error occurs and a developer does want to halt the program's execution?

2.4 The Panic System

Rust has a system to signal unrecoverable errors, called the panic system. This system is invoked using the macro `panic!()`. When invoked, the panic system halts execution of the current thread and unwinds its stack. As such, it should be used only for truly unrecoverable errors, such as violated invariants.

Some of Rust's built-in functions invoke the panic system. An example of this is `Result::unwrap()`. What this method does is return the value wrapped in `Ok`, or panic in case of `Err`. Essentially, it lets developers treat the `Result`'s error as unrecoverable.

```
let result: Result<i32, Error> = "1".parse();
let ok: i32 = result.unwrap();
```

Similarly, `Result::expect()` does the same, but it allows developers to specify a message the panic system should be invoked with, e.g. why they expected the result to not be an error.

```
let result: Result<i32, Error> = Ok(1);
let ok: i32 = result.expect("Some reason...");
```

These methods are useful but easy to abuse, as many errors are recoverable in some way. Using pattern matching to extract values instead is usually the preferred method.

How does this system work in a concurrent context? Consider the case where we spawn a thread which will return some value, e.g. of type `i32`.

```
let handle = std::thread::spawn(|| {return 1;});
```

To access this value, we call `JoinHandle::join`, which returns the type `Result<i32, Box<dyn Any + Send + 'static>>`.

```
let res = handle.join();
```

This blocks the current thread until a returned value is available. During non-erroneous execution, join returns `Ok`, with its wrapped value being the returned value. However, in case the spawned thread panicked, join returns `Err`, with its wrapped value being the argument given to the panic invocation. This makes it possible to handle panics of other threads.

It is possible to catch panics within the current thread using `std::panic::catch_unwind`, which takes in a closure to be executed. If a panic occurs while executing said closure, this function returns `Result::Err`, just like for `JoinHandle::join`. Catching panics this way is generally not recommended, as developers should use `Results` for recoverable errors instead.

2.5 The Rust Compiler

Many of Rust's safety features are enforced at compile-time. Because of this, the Rust compiler, *Rustc*, has to do a lot of work. Compilation is separated into discrete steps which have different associated intermediate representations (IRs), each optimized for a specific purpose. We will consider the function of each IR.

The first steps Rustc undertakes are similar to other compilers, which are lexing and parsing, which results in an Abstract Syntax Tree (AST). The compiler lowers this AST into different IRs through several stages. First, the AST is lowered to the High-level IR (HIR). The HIR is used to do type inference, trait solving, and type checking. With this information, it is lowered to the Typed High-level IR (THIR), which is used for pattern and exhaustiveness checking. The THIR is fully typed and slightly more desugared than the HIR. This THIR is lowered to the Mid-level IR (MIR). The MIR functions like a Control-Flow Graph (CFG) and shows the basic blocks of the program, along with how the control flow can go between them. It is used for borrow checking, data-flow based checks, as well as for many optimizations. Finally, the MIR is lowered to an LLVM-IR, which can be passed onto LLVM for code generation [4].

3 RELATED WORK

A lot of research has been done in the field of exception handling. This ranges from conceptual work forming the basis of modern exception handling, to field studies on how well exceptions are handled in practice. We will give an overview of such related work, including similar static analysis tools made for Java, an analysis of an embedded system written in C, an analysis of C++ exception handling constructs, and finally an analysis of Swift error handling.

3.1 Java Exception Flow Analysis

In their 2003 paper, Robillard and Murphy [10] detail a model for static analysis of exception flow in object-oriented programming languages. This model consists of three separate functions:

- encounters(s) → E
- catches(g, c) → E
- uncaught(g) → E

These refer to the exceptions encountered within a scope, the exceptions caught within a guarded scope, and the exceptions not caught within a guarded scope respectively. Together, these functions can model the exception flow within a Java program.

Furthermore, the paper details Jex, a static analysis tool using the model. This tool served as the main inspiration for our own tool. It takes in a Java program and maps out the exception propagation paths within it. Through visualizations, the tool helps developers understand what happens to exceptions, helping them identify potential issues.

3.2 Java Exception Handling Analysis

Likewise, in a 2007 paper, C. Fu and B.G. Ryder [5] detail their static analysis tool, also made for Java programs. It analyzes both exception flow within the program, as well as the exception handling code. It has a focus on analyzing highly layered programs such as servers, and how exceptions flow between said layers. With this, they can detect things such as catch-clauses that rethrow exceptions or ignore the exception altogether. It serves as a reminder that, even if explicitly caught, this does not mean an exception is handled properly.

3.3 C Error Analysis

In their 2006 paper, M. Bruntink, A. van Deursen, and T. Tourwé [2] analyze an embedded system, written in C and developed by ASML, for faults in its error handling. The program was written in C, a language without a specific error handling system, similar to Rust, but without conventions for defining and signaling errors, unlike Rust. Because of this, the developers had to define their own exception representation, complicating its analysis. The authors of the paper created a tool to analyze the system for faults in error handling. For this, they had to define both how errors are handled, and what kinds of faults could exist in the system. It is possible their definitions lead to both false positives (the tool flags some code as a fault, when it is not) and false negatives (the tool misses a fault).

3.4 C++ Exception Handling Constructs

Furthermore, a 2015 paper [1] examines exception handling constructs in C++ using both static analysis as well as a developer survey. The paper gives insight into how such constructs are used, as well as how developers think about using them. It reveals that, among other reasons, respondents to the survey avoid using exception handling constructs due to a lack of expertise to design an exception handling strategy. As such, it might be reasonable to a assume that a simpler error system would lead to developers more easily defining error handling strategies.

3.5 Swift Error Handling

Like Rust, Swift allows developers to use enumerations as errors and lets them use some pattern matching features when handling errors. However, unlike Rust, Swift error handling uses a try-catch system, and uses inheritance to define error types. A 2018 paper [3] examines how developers use Swift error handling in practice. Here, the authors found that developers are hesitant to define their own error types, and often use generic catch handlers. Furthermore, only 24.70% of developers use enumeration error types. Often-cited reasons for difficulties in using Swift's error handling features are confusion for novice developers, and technical limitations for experienced developers. This seems to corroborate that a simple but powerful error handling system might yield a better developer experience.

4 ANALYSIS DEFINITIONS & APPROACH

With the previously gathered information, we can start to define our own model for Rust error propagation. This section will cover the definitions needed to extract the error propagation from a program. We start by defining what an error is, go on to the ways these errors can be created and propagated, and end with what constitutes error handling. These concepts can then be used to implement our tool.

4.1 Error Definition & Creation

First of all, what are errors within Rust programs? As mentioned in section 2, errors are wrapped in the Result type. As such, we define an error as *any type used as the second argument of a Result.* Furthermore, the creation of an error happens whenever instantiating a `Result::Err` value. Note that this means a type can be used both as an error, and as a regular type. For example, in the following snippet, MyError constitutes an error when returned by `error_function`, because it is wrapped in a Result. It does not constitute an error when returned by `other_function`.

```
struct MyError;

fn error_function() -> Result<(), MyError> {
    return Err(MyError);
}
fn other_function() -> MyError {
    return MyError;
}
```

4.2 Propagation

Next, what does it mean to propagate an error? To cover the obvious case, an error is propagated by directly returning a Result, such as in the following example:

```
fn return_error() -> Result<(), MyError> {
    return error_function();
}
```

Additionally, as explained in section 2, another way to propagate errors is using the try operator, such as in the following example:

```
fn try_error() -> Result<(), MyError> {
    error_function()?;

    return Ok(());
}
```

As such, we define error propagation as *returning a Result, or using the try operator on a Result.*

4.3 Handling

Finally, what constitutes the handling of an error? In the context of propagation chains, the handling of an error happens at the end of such a chain. In other words, the handling of an error happens wherever an error is not propagated any further, such as in `handle_error()` in the following example:

```
fn handle_error() {
    let res = error_function().unwrap();
}
```

As such, we can define the handling of an error as *the point where an error is not propagated any further.*

4.4 Tool Usage

We have used these definitions to implement our tool, as detailed in the next section. This tool can be invoked using the bundled batch file `static-result-analyzer.bat`. This batch file ensures the needed dependencies are installed, and then configures and runs the tool. The tool can output either a call graph or an error propagation chain graph of the program it is run on.

Let us consider the following program:

```
fn main() {
    let x = propagate().unwrap();
}
fn propagate() -> Result<i32, MyError> {
    let x = result1()?;
    let y = result2().unwrap();
    return result3();
}
fn result1() -> Result<i32, MyError> {
    Ok(1)
}
fn result2() -> Result<i32, MyError> {
    Err(MyError)
}
fn result3() -> Result<i32, MyError> {
    Ok(3)
}
#[derive(Debug)]
struct MyError;
```

We used our tool to produce a call graph, as can be seen in figure 1, and a error propagation chain graph, as can be seen in figure 2.

The call graph shows that the function `main` calls the functions `propagate` and `Result::unwrap`. Then, `propagate` calls `result1`, `result2`, `result3`, and `Try::branch` (due to the try operator). Finally, `result1` and `result3` call `Ok`, while `result2` calls `Err` and the constructor of `MyError`.

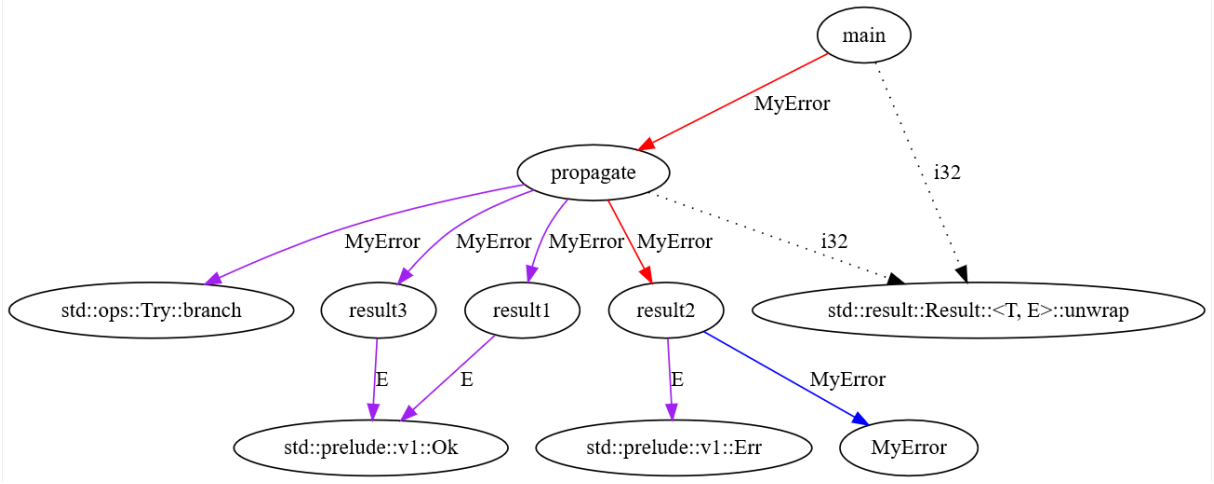

Fig. 1. Example Call Graph

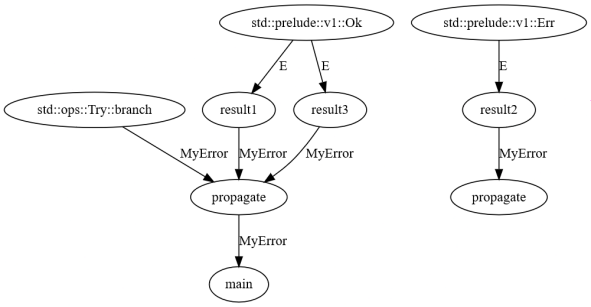

Fig. 2. Example Propagation Chain Graph

The edges are labeled with the return types of the called functions. Furthermore, they are colored according to our propagation and error definitions. Blue edges mean propagation of the returned value occurs (e.g. it is returned or used with a try operator), and red edges mean the return type is a Result. Having these properties together makes an edge purple. Having neither property makes an edge dotted, as they are of no real significance to the error propagation within the program.

The error propagation chain graph shows us how errors flow within the program. There are 2 separate chains, as errors are handled in 2 places: `main` and `propagate`. The error handler in `main` handles the errors originating from `result1` and `result3`, which then flow through `propagate`, and end up in `main`. The error handler in `propagate` handles the error originating from `result2`, which does not flow through any other functions. All errors are of type `MyError`.

Finally, our tool extracted some metrics from the program, namely:

- the number of chains: 2;
- the size of the largest chain: 6;
- the depth of the longest chain: 3;
- the average chain size: 4.

These metrics are useful for quickly identifying issues within a program, such as error paths that are too long.

4.5 Limitations

Our model comes with some limitations. For example, our definition of an error being handled is flawed. It assumes that if an error is not propagated any further, it is handled. This means that when an error is ignored altogether, our model still flags it as handled.

Now, let us consider the details of our implementation.

5 IMPLEMENTATION DETAILS

In this section, we cover some details of how we implemented our static analysis tool. We start by explaining how we use the Rust compiler to access its intermediate representations, then we describe how we create a call graph from the IRs, how we convert the call graph to distinct error propagation chains, and finally what specific metrics we extract from the propagation chains.

5.1 Compiler Invocation

The Rust compiler can be invoked programmatically, allowing developers to define callbacks to be called at specific points during compilation. However, this invocation needs to be configured properly, for example to link dependencies. We approached this by first running `cargo build -v` which builds the entire program, including its dependencies. The flag `-v` makes the output verbose, which makes it output the command used to compile the final program. From this invocation, we extract the arguments used, and use them for our own invocation.

Next, we use the `after_crate_root_parsing` callback which, as the name suggests, is called after parsing the crate root. From here, we have access to the intermediate representations, and can use them to analyze the program.

5.2 Call Graph

Now that we can access different IRs, we have to consider which one to use. We decided to use the High-level IR to create the call graph, as this IR is relatively close to the source while containing some useful extra information. Here, we start by finding the program entry point. From there, we walk the IR's tree, extracting all function and method calls. We store these calls in a graph, where functions are nodes, and function calls are edges.

We add extra information to this call graph such as whether a function call's return value is propagated (e.g. it appears in a return statement or with a try operator). Furthermore, we store return type information for each edge. We extract these types from the Mid-level IR when available, as this IR is fully typed. Otherwise, we use the HIR.

5.3 Propagation Chains

From this call graph, we can extract error propagation chains. These chains should show all paths an error can take up until the place it is handled. As such, we look for the function calls that can return an error, but do not propagate said error. For each of these edges, we create a chain in a new graph. Note that this means a propagation chain can contain multiple separate error paths. From there we compile all function calls from the called node which can return errors that are propagated. This graph is the default output of our tool.

5.4 Limitations

Our analysis tool does come with some limitations. For example, due to our method of extracting the compiler arguments, the tool only works on Rust programs that can be built using `cargo build`.

Furthermore, our method of extracting type information of function calls is imperfect, as the MIR is not always available and type information can not always be extracted. In such cases, we use the type information available in the HIR, which might still contain generic types (e.g. the type `E` for calls to `std::prelude::v1::Err`).

Also, the call graph is not complete, as it is impossible to create a call graph for used libraries. This means that function calls to a program from within a used library (such as when using a callback) will not appear in the call graph, and will thus not be considered for error propagation analysis.

With that, we can use our tool to analyze some open-source Rust programs.

6 CASE STUDY

In this section, we conduct a short case study on two Rust programs, Bottom[1] and Ferium[2]. Bottom is a customizable cross-platform graphical process/system monitor for the terminal, and Ferium is a fast and feature rich CLI program for downloading and updating Minecraft mods. These programs were chosen because they are open-source, actively maintained, and relatively popular (both have 1000+ stars on GitHub). Furthermore, they are of a sufficient size to potentially provide interesting insights into how Rust developers use errors within their programs. Specifically, Bottom consists of `25,515` lines of code, while Ferium consists of `2,578` lines. In this case study, we aim to answer the following question:

- **RQ1: What are the error handling strategies employed in the Rust programs Bottom and Ferium?**

To support our answer to this question, we also seek to answer the following questions:

- **RQ1.1: What is the average size of error propagation chains within Bottom and Ferium?**
- **RQ1.2: What are the largest error propagation chains within Bottom and Ferium?**

[1]https://github.com/ClementTsang/bottom/
[2]https://github.com/gorilla-devs/ferium

To answer these questions, we will use our tool to analyze the programs. Its output will be used to get an overview of the propagation chains within the program. Using this as a guide, we manually analyze the program's source code to find out what error handling strategies are used.

6.1 Results

We have compiled the metrics extracted by our analysis tool in table 1. We go over the results in some detail for each program individually.

6.1.1 Bottom. Our tool tells us there are `32` separate places where errors are handled within Bottom. In the propagation chain graph we see these places are:

- `options::colours::ConfigColours::is_empty`
- `options::get_use_battery` (2x)
- `data_collection::disks::windows::get_disk_usage` (3x)
- `data_collection::disks::windows::bindings::volume_name_from_mount`
- `data_collection::disks::windows::bindings::all_volume_io` (6x)
- `data_collection::disks::windows::bindings::volume_io` (2x)
- `data_collection::disks::windows::bindings::close_find_handle`
- `data_collection::DataCollector::init` (2x)
- `panic_hook` (7x)
- `main` (3x)
- `canvas::components::data_table::DataTable::increment_position` (2x)
- `widgets::process_table::ProcWidgetState::update_query`
- `event::handle_key_event_or_break`

Of these, the largest propagation chain consists of `73` function calls, which ends up in `update_query`. This suggests this propagation chain has very complex behavior. Indeed, it deals with parsing queries, which have a complex rule set, including RegEx support. This complex rule set means many opportunities for parsing a query to fail, resulting in an error. Examining the propagation chain, we find that there are only `3` different error types that can occur here:

- `utils::error::BottomError`
- `regex::Error`
- `humantime::DurationError`

Here, `utils::error::BottomError` is a self-defined enumeration type with different specific errors. It is extensively used within the program for most errors. This indicates the developers are familiar with Rust's error features and make good use of them. The handling of this error occurs with the use of pattern matching, where the error message is made ready to be viewed by the end-user within the program.

The longest path an error can take within the program before it is handled is `6` chained function calls. Additionally, the average propagation chain consists of `3.75` function calls. This means errors are handled soon after they come into existence, and tells us that most error handling code handles specific errors.

Finally, we run `cargo check` on the program. This would alert us of ignored `Results` as the type is flagged with `#[must_use]`.

Program	Number of Chains	Largest Chain Size	Longest Chain Depth	Average Chain Size
Bottom	32	73	6	3.75
Ferium	11	17	3	2.64

Table 1. Metrics Extracted from Bottom and Ferium

We find no such ignored `Results`, and as such we can conclude no errors are ignored within the program.

6.1.2 Ferium. Using our tool to analyze Ferium, we find there are 11 different places errors are handled within the program. These places are:

- `actual_main` (3x)
- `subcommands::modpack::upgrade::upgrade`
- `download::clean`
- `download::download`
- `subcommands::profile::create::create`
- `subcommands::upgrade ::get_platform_downloadables` (2x)
- `main` (2x)

The largest propagation chain consists of 17 function calls. It is shown in figure 3. This indicates that the end of this chain, `actual_main`, can have many different error types flowing to it. Examining this chain closely, we find that the following error types can flow here before being converted to the final type, `anyhow::Error`:

- `anyhow::Error`
- `fs_extra::error::Error`
- `std::io::Error`
- `dialoger::Error`
- `libium::modpack::add::Error`

As such, this error handling code can deal with many different errors. Analyzing the program's source code, we see that this propagation chain deals with executing a sub-command to add a modpack to the users current selection of mods. This is a complex feature where many things might go wrong, explaining the many error types. Furthermore, we find that these errors are treated as unrecoverable by exiting the program with an error code. The program is designed to be invoked with a sub-command from the command-line, where executing said sub-command is the only task of the program. As such, it makes sense to halt execution if the program fails to execute part of this sub-command.

Moving on, the longest single chain consists of only 3 chained function calls. This means there are no errors that take long paths within the program before they get handled, indicating a good error-handling strategy.

The average chain size is `2.64`, which indicates that most error handling code is specialized for a single error, as there is not much room for error conversions in short propagation paths.

Finally, we run `cargo check` again. This does not warn us of any ignored `Results`, so we conclude there are no errors which are ignored.

6.2 Conclusions

With these results, we can answer our posed research questions:

6.2.1 RQ1.1: What is the average size of error propagation chains within Bottom and Ferium? We found that the average size of error propagation chains within Bottom was `3.75`, while in Ferium it was `2.64`. As discussed, this indicates most of both programs' error handlers handle specific errors. Furthermore, it indicates most errors are handled swiftly after they come into existence.

6.2.2 RQ1.2: What are the largest error propagation chains within Bottom and Ferium? For Bottom, we found that the largest error propagation chain ended in the function `update_query`, consisting of 73 function calls. This large propagation chain deals with parsing a query, with many rules that might be violated, causing errors.

We found that the largest error propagation chain in Ferium was one that ended in the function `actual_main`. It consists of 17 function calls. It was this sizable because it belongs to a complex feature, adding a modpack, whose errors are treated as unrecoverable.

6.2.3 RQ1: What are the error handling strategies employed in the Rust programs Bottom and Ferium? Bottom handles errors in a way where end-users get feedback about what went wrong within the program, without halting it. As a graphical program with many different components, it prioritizes up-time of the overall system. Furthermore, it makes great use of self-defined error types.

Ferium, as a CLI program made to execute a specific task each time it is ran, treats many errors as unrecoverable. This fits its use-case, as an error while executing that task means it cannot proceed, and should halt execution.

7 CONCLUSIONS

In this paper, we have looked at how Rust allows developers to deal with errors in their programs. Furthermore, we have done a case study on what strategies developers use within two open-source programs. With the knowledge obtained, we can begin to form an answer on our research question:

7.1 In what ways are errors created, handled, and propagated with the Rust programming language?

We found that errors are created by instantiating a `Result`, specifically `Result::Err`. This wraps the actual error value, allowing developers to use any type as an error. For example, developers might define their own enumeration type with many specific error values to indicate errors within their program.

Furthermore, errors can be handled in any way the developer likes. Rust provides many pattern-matching features to support this. They can be treated as unrecoverable errors by leveraging the panic system, for example by using `Result::expect()`.

Finally, errors from called function can be propagated by either returning them from the function, or by using the try operator. The try operator might convert error types using the `From` trait.

7.2 Case Study

In our case study, we used our tool to analyze the usage of Rust's error system within two open-source programs. We showed how to interpret the extracted metrics, as well as the viability of using

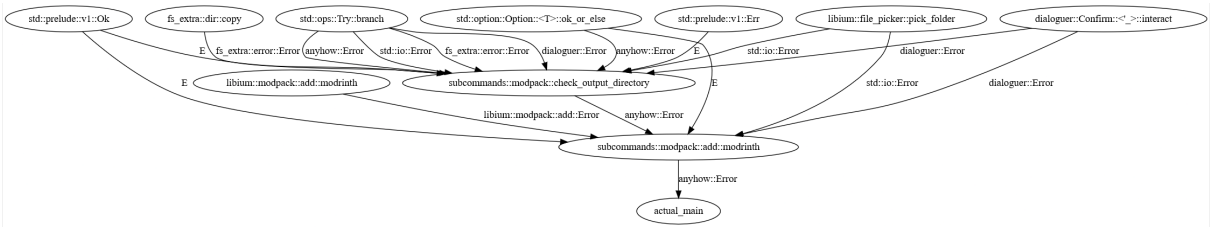

Fig. 3. The Largest Propagation Chain in Ferium

our tool for such a use-case. As such, we feel our tool provides a great stepping stone for further research on Rust's error system.

8 FUTURE WORK SUGGESTIONS

The aim of this paper is to serve as a first step towards understanding Rust's error system, and how it is used in practice. As such, there are many potential steps still to be taken and topics yet to be covered.

For example, while our tool can be very useful in its current state, its usefulness can be improved. One of the features that did not get implemented is showing where the panic system is used within a program. Since usage of this system relates to unrecoverable errors, this information would be very useful to developers. We left this feature out due to time constraints and because there already exist separate tools to do this.

Furthermore, the way our tool invokes the Rust compiler might be altered to allow for usage on more complex programs. Our current method restricts usage to programs that can be built with just the command `cargo build`. Improving this might allow for insights into the ways developers use and handle errors within more complex programs.

Additionally, the tool could extract more metrics from the propagation graph, extending its usefulness to developers. For example, it could count the number of different error types within each chain.

Finally, we suggest our tool be used on more and larger programs. Our case study of two programs was limited in its scope due to time constraints. A larger study would lead to better insights into how developers use and handle errors within Rust programs.

ACKNOWLEDGMENTS

I would like to thank my supervisor, professor Fernando Castor, for his insight and guidance. He was instrumental in making this research project a success.

REFERENCES

[1] Rodrigo Bonifacio, Fausto Carvalho, Guilherme N Ramos, Uira Kulesza, and Roberta Coelho. 2015. The use of C++ exception handling constructs: A comprehensive study. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (Bremen, Germany). IEEE.

[2] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. 2006. Discovering faults in idiom-based exception handling. In *Proceedings of the 28th international conference on Software engineering* (Shanghai China). ACM, New York, NY, USA.

[3] Nathan Cassee, Gustavo Pinto, Fernando Castor, and Alexander Serebrenik. 2018. How swift developers handle errors. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg Sweden). ACM, New York, NY, USA.

[4] The Rust Community. n.d.. *Overview of the compiler.* Retrieved April 30, 2024 from https://rustc-dev-guide.rust-lang.org/overview.html

[5] Chen Fu and Barbara G Ryder. 2007. Exception-chain analysis: Revealing exception handling architecture in java server applications. In *29th International Conference on Software Engineering (ICSE'07)* (Minneapolis, MN, USA). IEEE.

[6] John B Goodenough. 1975. Exception handling. *Commun. ACM* 18, 12 (Dec. 1975), 683–696.

[7] Steve Klabnik, Carol Nichols, and the Rust Community. n.d.. *The Rust Programming Language.* Retrieved May 01, 2024 from https://doc.rust-lang.org/book/

[8] Stack Overflow. 2022. *2022 Developer Survey.* Retrieved June 10, 2024 from https://survey.stackoverflow.co/2022/

[9] Stack Overflow. 2023. *2023 Developer Survey.* Retrieved June 10, 2024 from https://survey.stackoverflow.co/2023/

[10] Martin P Robillard and Gail C Murphy. 2003. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.* 12, 2 (April 2003), 191–221.

[11] Yuchen Zhang, Yunhang Zhang, Georgios Portokalidis, and Jun Xu. 2022. Towards understanding the runtime performance of rust. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester MI USA). ACM, New York, NY, USA.