Measuring Code Modernity in Rust

CHRIS BLEEKER, University of Twente, The Netherlands

The measurement of modernity has been a common topic of discussion in many different fields of study. This also applies to measuring the modernity of source code within computer science. Previous studies have shown that the measurement of code modernity is possible using static analysis and can produce useful results, however these studies have only focused on a single strategy of arriving at this measurement. This research tries to find and explore the suitability of several different code metrics for measuring code modernity in the Rust language.

Additional Key Words and Phrases: Code Modernity, Modernity Signature, Rust, Static Analyisis, Source Code Analysis

1 INTRODUCTION

The subject of measuring code modernity has been an often studied topic in the last years, with multiple papers coming out discussing the measurement of this metric, such as the research conducted by van den Brink et al. [36] regarding measurement of code modernity in PHP using static analysis. There has also been discussion about the interpretation of the values presented, for example by Zubcu [39], who reasoned about the normalization of the modernity measure.

Determining a modernity signature of a code base can be useful for many applications, such as measuring how well maintained a project or how far a fork has strayed from its origin. It can also be considered as a part of a code quality measure, and play a crucial role in reviewing contributions and keeping track of large code bases to make sure they meet the latest quality standards and adhere to the best practices set by the language.

In most papers discussing this topic, the measure of code modernity is considered a measure of either the age of a project [6], or a measure of the amount of contemporary ‘modern‘ features in the code used from the respective programming language [39], leaving a lot of room for interpretation.

Similar research has been done on a variety of different languages besides Python and PHP, such as Güdücü’s research on automatically constructing weighted abstract syntax trees in Java [17]. However, the concept of modernity could be very different between established, slow-evolving languages such as Java and Python, and newer, more cutting-edge languages such as Rust. Besides this, no requirement for a user-installed run-time executor in Rust could entice developers to adopt newer features more quickly.

Rust has grown in popularity significantly over the last few years, and is slowly starting to be used in production more often at a variety of companies. While static analysis tools for Rust exist, the ecosystem is still growing and there are a lot of gaps left open, such as the one we address in this paper.

TScIT 40, February 2nd, 2024, Enschede, The Netherlands

2 PROBLEM STATEMENT

Looking at previous implementations of code modernity measures, all of them use some sort of static analysis technique to version parts of the code. For example, van den Brink et al. [36] used weighted abstract syntax trees, which are then collected and assigned versions. From these version numbers, an n-tuple is made counting the amount of features used from each version, which is then plotted to visualize the results.

In a similar fashion, Admiraal [6] used a program called VERMIN [19] to version language items, resulting in comparable results based on n-tuples of versions. This begs the question if analyzing code version distribution is the only way to achieve such a measure.

It seems to be implied that a measure of code modernity is equal to the amount of features used from each version. However, the fundamentals are never looked at closely, and other metrics for measuring this are never considered.

This problem statement can be summarized in the following research question:

What metrics are suitable to be used for measuring code modernity in the Rust language?

To aid in answering the research question, the following sub-questions will also be considered:

(1) What is a good definition of code modernity?
(2) What are some of the consensuses in the community of what constitutes idiomatic Rust code?
(3) Can existing static analysis tools be used to aid in measuring code modernity?

3 RELATED WORK

Several previous implementations of a code modernity measure have been made in the past. Van den Brink et al. in their research has done this using weighted abstract syntax trees to version parts of PHP source code [36]. Admiraal later did something similar in Python, using language items to version source code. These results have later been analyzed by Zubcu [39] to study the effect of various normalization techniques to visualize the data obtained by these studies. The results of this study can be used to construct a better metric, and find ways in which different metrics can be used to measure code modernity.

On the topic of idiomatic Rust code, various studies have been done analyzing common Rust patterns and their effectiveness, such as the one conducted by Qin et al. studying the use and effect of safety guarantees set by Rust and the `unsafe` block [27]. Additionally, Li et al. developed the program MIRCHECKER [22], which uses static analysis to detect bugs in Rust projects using Rust’s Mid-level Intermediate Representation (MIR). Tools such as Clippy [25] contain a growing number of community accepted lints, which is widely used across Rust projects to analyze both behaviour and style. Rust also has well-established patterns that are widely referred to and used, which are documented in various places such as the Rust-by-Example book [32].

Similarly, outside of the Rust realm, Farooq and Zaytsev studied what makes Python code idiomatic [16]. Allamanis and Sutton also presented a method to mine idioms from existing code in an automatable fashion [8]. There has also been a significant amount of research done in the area of using machine learning to perform static analysis, such as Dewangan et al. training a model to be highly effective at detecting code smells [14]. Sandouka and Aljamaan, in a similar fashion, created a large dataset of 2000 samples, to use in training a model to detect code smells in Python [29]. While these all perform static analysis in some form to find certain pattern in code, it seems to be primarily for the purpose of finding problems that need to be solved, instead of classifying code bases as this paper aims to do.

4 METHODOLOGY

The general process of this research can be divided into three parts, roughly corresponding to one part per sub-question:

(1) Finding a good definition of code modernity with few compromises
(2) Reviewing literature to find potentially viable metrics for measuring code modernity
(3) Implementing these metrics and verifying their suitability

The first metric to be implemented will be comparable to the methods used in Admiraal's [6] and van den Brink et al.'s [36] research, to act as a baseline. This method has been proven to be successful for other languages, and if successful, can act as a good basis of comparison for other metrics to determine their suitability for measuring code modernity.

4.1 Definition of code modernity

For our definition of code modernity, existing definitions should be taken into consideration. Besides looking in the field of computer science, other fields of study should also be considered to see if there is a universally applicable definition of modernity and if this can be adapted to fit our needs. It should take into account the more general definitions of modernity and modern, and build on that in the context of programming languages and code.

This definition should be able to encompass and not invalidate previous research on this topic, and provide room for further evolution of the concept of code modernity in the future. The definition will therefore be compared to the various papers discussed previously to make sure of this.

The concluding definition that is achieved in this step will be used for the rest of the research.

4.2 Literature review

In this part, a list of potentially viable metrics should be constructed. Depending on the final definition from the previous part, this most likely involves a search for various properties of source code that have changed over time, thus contributing to code modernity. This can include ‘hard‘ properties such as minimum versions, which has a single set value per code base with no room for variation, and ‘soft‘ properties such as idioms, which can differ based on the selection of a subset of idioms, and the chosen implementation.

To find these metrics, we can use previous research such as the ones mentioned in Chapter 3. We can also look for community consensus by using popular rust idioms and style guides such as the Rust Unofficial Design Patterns guide [31].

To find any consensus within the Rust community around Rust idioms, it should be properly defined what the community is in the context of this research paper. For a place to be called a part of the community, it should either be linked to often by official resources such as the official Rust website, or have an audience that is sizable and not focused on a particular niche, rather on the language as a whole.

4.3 Implementation and analysis

To verify the suitability of the properties obtained, a prototype measuring these properties should be implemented and tested. The resulting program can then be used to analyse, measure and verify these metrics over a collection of known Rust projects. This collection consists of multiple crates from the crates.io repository, namely `actix-web`, `regex`, `async_std`, `tokio`, `syn`, `bevy_ecs`, `serde_json`, `clap`, `egui`, `axum` and `ratatui`. This collection of crates includes crates from various different ecosystems, such as asynchronous runtimes, web services and game development. It also includes projects created at various points in time, which could influence the modernity that the code base started at. All of the selected crates are well maintained and have been for most of their history, which should indicate result in the modernity gradually increasing over time as new code gets added, and older code gets rewritten to be more modern.

While these crates have been selected to cover a wide range of Rust projects, and to cover most features and idioms commonly seen in their ecosystem, the small-scale nature of this dataset means that they cannot represent all Rust code, even in their own niche. The dataset is deliberately kept small as to limit the scope of the research, but should still be general enough to show if metrics are suitable to be used to measure code modernity beyond chance.

After these results have been acquired, different metrics can be put against each other and evaluated to see if they show the same data as each other. The language item version metric can also be compared to the similar metrics in previous research papers that measure code modernity [6, 36].

To determine if a metric is suitable to be used for measuring code modernity, we will be considering two use cases. Firstly, we will look if a metric can differentiate between different crates. For this, if there is a significant difference between two crates, then there should be a reason for this difference. For example, we expect `axum` to be of higher modernity than `async_std`, as the former has been developed more recently and, as determined by manual review, uses more modern code than the latter.

Secondly, we will look if a metric can differentiate between versions of the same crate. This can generally be done by looking at trends over time. As all selected crates are well maintained, it is expected that the modernity goes up over time, and thus a trend either up or down can indicate a suitable metric. While this is not conclusive evidence, as there are different reasons a trend can exist over time, combined with reasoning on why this trend exists, it can provide a good indication.

5 METRIC ANALYSIS

5.1 Definition of code modernity

Between the discussed published papers about code modernity, there seems to be no one universal definition of code modernity. Van den Brink et al., in their paper about modernity measures in PHP, does not mention any definition of code modernity itself, rather only looking at it as a measure, which is parallel to the amount of versioned items used in source code [36]. He does use this statistic to (partially) determine the age of a code base. Expanding on this, Admiraal later defines code modernity as "a scale of measuring the age of a project" [6]. Earlier in his paper, he also mentions how pythonic idioms can affect code modernity, however this is not part of his definition. Zubcu, in his research about normalization of code modernity measures, notes that "modernity represents the extent to which a software system's source code leverages the contemporary features and capabilities inherent in the respective programming language," [39].

Another field of study where modernity is an important subject is sociology, where Shilliam defines it as "a condition of social existence that is significantly different to all past forms of human experience," [30]. In the context of software, this puts a large emphasis on age and significant differences between two code bases with different modernity signatures.

The Cambridge Dictionary defines modernity as "the condition of being modern" [13], where modern is defined as "designed and made using the most recent ideas and methods," [12]. The Oxford English Dictionary agrees on this definition for modernity [24], however defines modern as "being in existence at this time; current, present", again bringing back a strong emphasis on age [23].

To conclude, almost all definitions presented refer to code modernity as a measure of age in some way. However, defining modernity solely as a measure of age is ambiguous and conflicts with some of the other editions presented. Particularly, this definition seems to leave no room for any external influences to the measure, and disregards code quality entirely. A lot of weight is put on recent ideas and current idioms, which should also be reflected in the final measure. A new definition should take both of these approaches to modernity into account.

In this paper, we define code modernity as **a measure that represents the point in time that the code would've been written when using the contemporary features and idioms of that time**.

This definition still works well retroactively and can encapsulate the results of the previously discussed papers on this topic. It still represents a measure of age, but alongside a clause that takes code quality into account and allows code bases to change and evolve over time. Both Admiraal and van den Brink et al. [6, 36] used an n-tuple of versions to represent the measure, which can be seen as an almost exact translation of the definition we created.

5.2 Measure Metrics

Past research has shown that analysis of versions of language items can effectively predict differences in relative age of code bases [6, 17, 36]. This method is also replicated for this research. The resulting metric can be used to verify if this method holds up for code written in the Rust language, and can be referred to in the same context alongside the results found for Python and PHP.

To get an idea of the general consensus of the Rust community around what is considered idiomatic code, we first need to define what we consider to be the Rust community. We aim to find the opinion of the majority of the community, and thus our main target is the larger hubs of discussion surrounding the programming language. These should be easy to find and easily accessible, for example by being linked from the official Rust site. They should also be reputable, for example by being endorsed by the core team behind Rust. As we aim to explore the general shared of the developers using the language, and not necessarily the theoretical best practices, this paper places a greater emphasis on online discussion platforms, surveys conducted on developers and widely-used code bases, rather than literature curated by a limited number of authors.

In this paper, the community includes the Rust subreddit, as it acts as a major hub for Rust discussion, and is relatively easy to discover, as indicated by the amount of beginner questions posted [5]. The official Rust users forum [4] is also relevant, as a central place that is linked from the official site and endorsed by the team behind the language itself [1]. Similarly, the official Rust blog invites everyone yearly to fill out a survey on the Rust language [3], which is also referenced in this paper. Lastly, we also consider various well-known projects on GitHub and the official package repository crates.io to be reflective of the community.

While a vocal part of the Rust community recommends against immediately using new Rust features, instead recommend to wait a few months to ensure everyone updates their compiler, there is generally no reason to stay far behind [18, 20]. A survey conducted in 2018 by the Rust Survey Team has shown that almost 88% of respondents were using either the current stable release or the latest nightly [34]. Additionally, according to the stats tracked by lib.rs, we can see that in the 4000 most recently updated crates as of November 2023, over 50% is currently incompatible with release 1.63, which was released in August 2022, compared to 13% of all crates published to the official repository [21, 33]. This gives us good reason to believe that new releases target more recent versions of the language, and thus that versions released in the past year are more likely to be used.

A similar metric to versioned language items is the Minimum Supported Rust Version (MSRV) of a code base. This refers to the minimum version of the Rust compiler that can still compile the project in full. This has the flaw that dependency MSRV values can bubble up without requiring any changes to the parent code base itself. However, it can still give us a good idea when looking at larger time spans. Besides the actual MSRV, Rust reserves a field in the project manifest file to provide a reported MSRV value, to be kept up to date by the maintainer [7]. While project maintenance does not necessarily always need to lead to an increase in code modernity, a change in reported MSRV could imply this. The used Rust edition of a code base can signify the same thing in more projects, as it is mandatory to include this field [7]. However, as there are only 3 possible values for the edition, it can most likely not be used on its own.

Aside from changes in versions, the way the Rust language is used also changes over time. As new features get introduced, and flaws in old patterns get discovered, the recommended way of using the language changes. This can be seen in the changes in idioms over time. As we have defined idioms to be a part of the definition of code modernity, accurately measuring the usage of idioms can make a great candidate metric for measuring code modernity.

One idiom that is often discussed is the usage of unsafe Rust code. While the usage of unsafe code has always been discouraged in favour of safe abstractions, many high-profile projects still used a lot of unsafe code. A GitHub issue pointing this out in the web-framework `actix-web` [10] caused a domino effect in the community [15] to scrutinize every single use of unsafe code in code bases. Astrauskas et al. have remarked that in a 16 month period until November 2020, the amount of crates in the main repository crates.io using unsafe code had dropped from 29% to 23% [9]. According to lib.rs, around 10,000 new crates were published on crates.io in this 16 month period [21], which can indicate that newer code bases on average has less unsafe code than older code bases. This gives us good reason to believe that the relative amount of unsafe code in a code base could be a suitable metric to measure code modernity.

A second feature that is gaining a lot of popularity is the built-in async functionalty of Rust. In 2018, new syntax was introduced for asynchronous functions [38], which is nowadays commonly used by web frameworks and other backend applications. This is one of the most popular fields that Rust is used in, according to a survey conducted by the Rust Survey Team in 2021 [34]. This could be covered by the version signature metric, which includes asynchronous methods, but most of the commonly-used asynchronous APIs are not implemented in the standard library. For this reason it is a better idea to consider other metrics, such measuring the proportion of asynchronous functions to non-asynchronous functions in a code base.

Lastly, we can gauge how much external tools can help with measuring code modernity. A good candidate for this is Cargo Clippy [25], which is a tool that provides additional lints for Rust source code. Most of these lints are aimed at catching bugs and making code more idiomatic. It is generally considered good practice to use Clippy within the community [37], and while the documentation does mention that it is opinionated, and thus disabling lints is fine, in practice most lints are left enabled [26]. The project is maintained by the Rust team themselves, and the lints are discussed by the community before being implemented and stabilized. Lints that are particularly good often get promoted to the compiler itself. This all makes Clippy a reputable linter detecting common code smells agreed upon by the core Rust community. Integration with popular IDEs give good incentive for developers to bring the amount of warnings down to zero, and as lints are added with time, the amount of warnings produced by Clippy could give us a good indication of the age and the idiomatic nature of a code base, and thus code modernity.

6 IMPLEMENTATION

To aid in automatically measuring and analysing the metrics discussed, the Ruvolution tool [11] has been created as part of this research. This tool supports analysing any Rust code base that can compile with the locally installed Rust compiler. It also contains utilities to select and download different crate versions from the crates.io repository and analysing each version, outputting a CSV file to be used for analysis. This tool implements measuring a version signature, edition, reported MSRV, amount of Clippy warnings emitted, amount of expressions marked `unsafe` and amount of functions marked `async`, along with the total amount of expressions and functions.

Ruvolution is written in Rust to take advantage of the libraries already available to work with Rust source code, most importantly the `syn` crate, which is a Rust parser library. Before parsing any source code, we use the `cargo-expand` tool [35] to apply any compiler pre-processing and consolidate the source code into a single file. This comes with the downside that any macros are also expanded, leading to inaccurate results in some cases, however it saves a lot of work of re-implementing parts of the compiler in order to make the source code digestible.

For each crate analysed, all versions are fetched using the crates.io public API. From these versions, only stable releases in increasing `semver` order are retained, and any other versions are not considered. This is so that the resulting versions represent the versions used by most developers using this crate. After this, 20 versions are selected in even intervals and analysed.

To fit the scope of the research, some concessions had to be made in the exact methods used to measure the metrics. To calculate the version signature of a code base, first a version symbol map has to be made. The initial plan was to only analyse the `std` crate for this, however a lot of functionality is imported from the `core` and `alloc` crates. For this reason, these two crates are also analysed, and linked back to the `std` crate using import aliases, which is a rudimentary re-implementation of the module system found in the Rust compiler. This analysis could be made more reliable by finding a way to hook into the compiler, such as using the JSON output of the `rustdoc` documentation generator [28], but as of December 2023 there is no trivial way of doing this.

Then, for the target code base, the symbol map is used to count the versions used in imports and expressions. For simplicity, import aliases are not considered for the target code base. The resulting information is gathered into an n-tuple counting the amount of language items per version, similar to the research conducted by van den Brink et al. [36]. Then, to consolidate this information, the counts x_i are normalized using a log-normalization as to emphasize newer versions [39], and a weighted average s is taken of the versions v_i, as shown in eq. (1).

$$x'_i = \frac{\ln x_i}{\ln \prod_{i=1}^{m} x_i}, \quad s = \frac{\sum_{i=1}^{m} x'_i v_i}{\sum_{i=1}^{m} x'_i} \tag{1}$$

During this analysis, the program keeps track of additional stats in order to construct the other metrics, namely counting expressions and functions. This data is then used to calculate proportionally

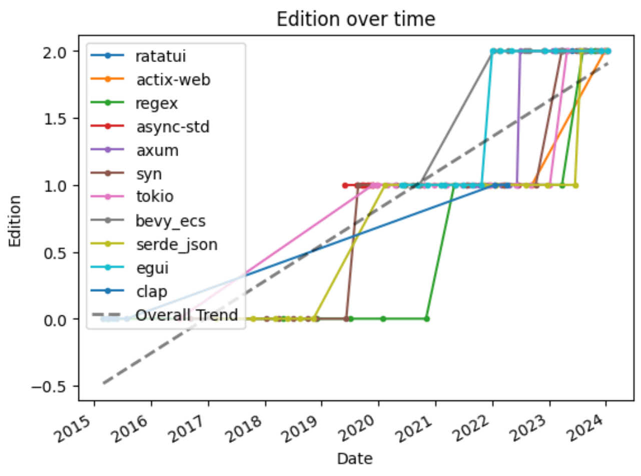

Fig. 1. Edition from `Cargo.toml` over time

how many `unsafe` expressions and `async` functions are in the target code base.

Finally, using the CLI interface of Cargo, it executes Clippy, which checks the code base and outputs the number of warnings emitted. This statistic is also recorded in the resulting CSV file. This is the most resource intensive part of the analysis, as it requires a full compilation of the code base, and there is unfortunately not much we can do to make this faster.

For manual analysis, the CSV files are processed using a Python notebook and plotted using `matplotlib`. While plotting, the program uses linear regression to calculate a trend across all gathered data, and additionally draws this in the plot. While this trend line can give us an insight, it should be used carefully, as it tells us nothing about reliability of the data.

7 RESULTS

The individual metrics are plotted against time to show the relation between the metric and modernity. Each line represents a single crate, with the points representing different versions. The gray striped line in each plot represents the general trend of the data.

The measurements were done using a local install of Rust 1.74.0 on Debian 13 running on WSL2. The versions considered for analysis are all released before January 19th, 2024.

For every analyzed crate, the edition noticeably goes up over time in fig. 1, with every version published after 2022 being on the second edition, and the majority transitioning to the third edition in 2023.

A similar result to the edition metric is found for the reported MSRV values in fig. 2. However, not all crates reported their MSRV in the manifest file, with only 8 out of 11 crates being shown in this plot.

The version signature plot in fig. 3 shows no clear trend across all crates. While some crates do show an upward or downward trend as time increases, others seem to have no clear direction. Between different crates there also seems to be a large variance in the version signature, but between versions there are generally only small variations, with a few exceptions.

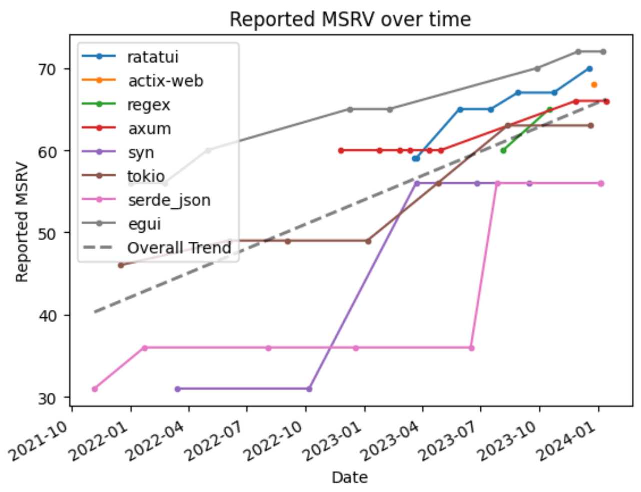

Fig. 2. Reported MSRV from `Cargo.toml` over time

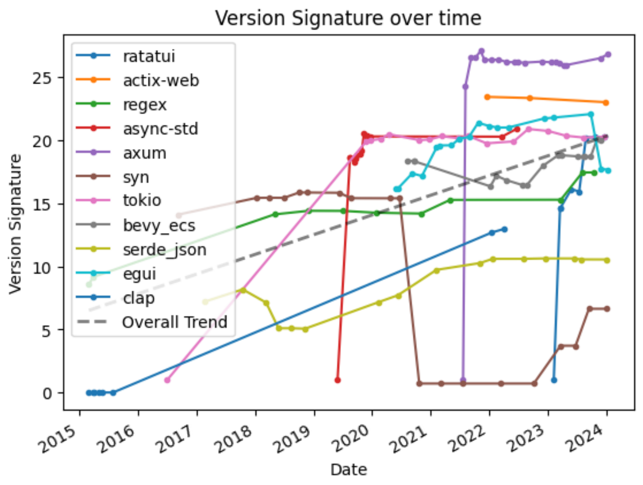

Fig. 3. Normalised version signature over time

The amount of unsafe expressions is proportionally shown in fig. 4 to the total amount of expressions. While most values hover around zero, it varies per crate, and there are some crates with peaks up to 40% unsafe expressions.

The proportion of async functions as shown in fig. 5 stays the same for most crates. More than half of the crates analysed do not contain any async functions, and thus are not included in this metric. There seems to be a sharp drop-off at the start for async-std, after which it stabilizes. Between different crates there seems to be no pattern in proportion to their expected modernity.

The amount of clippy warnings per expression as shown in fig. 6 shows a very clear trend towards zero as time goes on for all crates. While the metrics is very unstable at the start of the lifetime of most crates, it seems to stabilize as time goes on. Between crates there seems to be no relation.

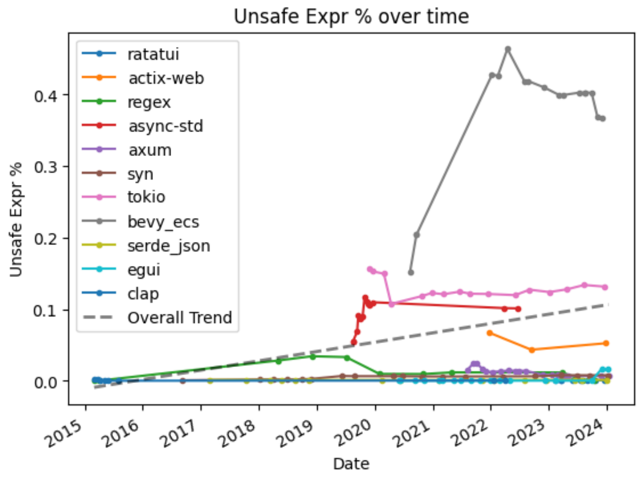

Fig. 4. Fraction of expressions marked `unsafe` over time

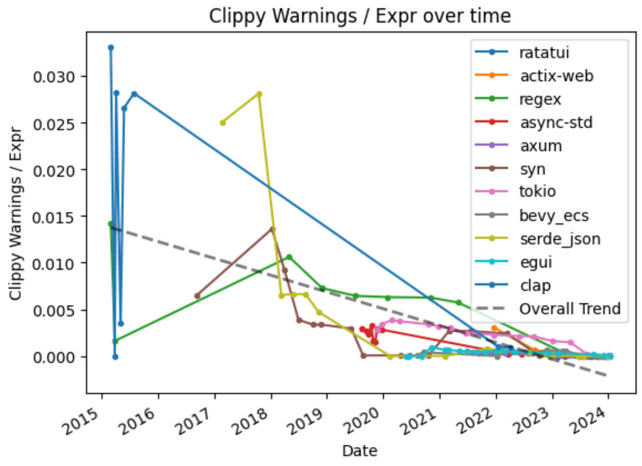

Fig. 6. Amount of reported Clippy warnings per expression over time

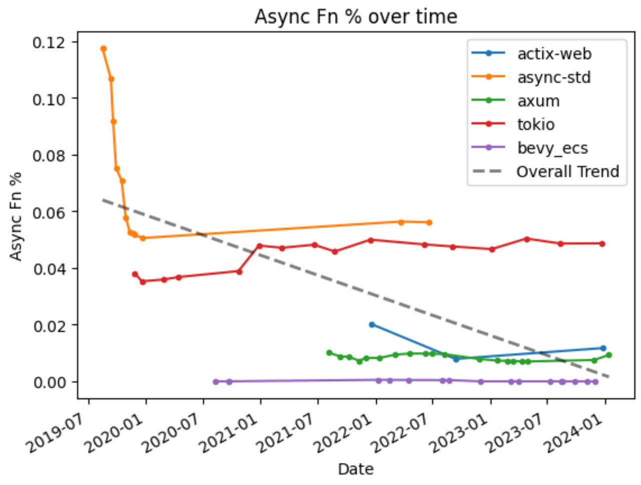

Fig. 5. Fraction of functions marked async over time

8 DISCUSSION

It is important to separate two use cases for a modernity measure, namely the evolution of code modernity within a code base, and comparison of modernity between different code bases. To determine suitability for measuring evolution of modernity, we should be looking for a trend as time goes on. For comparing different code bases, the relation between different lines should be considered.

Both the edition metric in Fig. 1 and the MSRV metric in Fig. 2 look to be very suitable for both use cases. As all selected crates are well-maintained, we expect the modernity to go up over time. Both of these plots show a very clear trend upwards as time goes on, as indicated by the trend line. The MSRV metric shows a couple outliers, namely `serde_json` and `syn`, which stay behind in minimum required version for a long time. Both of these crates are in the top 20 most downloaded crates according to crates.io as of January 2024, and thus have good reason to stay behind on minimum version to not break any legacy code bases. For this reason, pull requests using modern features can sometimes not be accepted, holding back the code modernity, and thus staying behind in this graph is consistent with our code modernity measure.

Edition is a required field in all Rust projects, and thus provides a very consistent, always present measure. However, the Rust edition has only been increased two times thus far, giving only three possible values for edition. While the upwards trend and the relation between lines makes this a very good measure, on its own its not enough to measure modernity accurately due to the limited amount of values.

The opposite is true for the MSRV field. As it has been introduced only recently [7], the metric only dates back to late 2021, and not all crates analysed use this field. However, for the crates that do, this metric shows a very clear trend upwards, with all lines ending in a higher version than they started at.

Looking at the version signature metric over time in Fig. 3, there is a slight upwards trend, however 4 out of the 11 crates do not follow this trend, ending at roughly the same version or lower than it started. This is despite all crates selected being well maintained, and thus the code modernity is expected to increase. For this reason, this metric is likely not suitable to draw any conclusions from, however the increasing trend does provide a compelling reason to do further research with this metric. An interesting outlier is the `syn` crate, which drops to almost zero late 2021, which was caused by a change in a widely used macro. Since analysis is done on expanded code after macros, this means that this single change was counted very heavily in the version signature, which is an unfortunate side effect of the implementation.

Comparing modernity between crates using the version signature seems possible, however not consistent. There seems to be a rough relationship between the time period most of the code base was written and the vertical position of the line. For example, `axum`, the top line, is a relatively new web framework, which was created in 2021. A lot lower is `serde_json`, which is a JSON serialization framework that has existed since 2015. While there might be a high level similarity between these two statistics, it can not be used when comparing two crates individually, as there is too much variation in version signature between crates of similar modernity.

Looking at the percentage of `unsafe` expressions over time in Fig. 4, we can see that most crates are staying very close to zero. A few crates stay high above zero, including `actix_web`, `async-std`, `tokio` and `bevy_ecs`. The common property between these crates are that they are all highly optimized code bases. This means that a lot of assumptions are made that are beyond the capabilities of the Rust compiler, thus unsafe is used. This does mean that these assumptions need to be kept track of by the developers, which undermines one of the core principles of the Rust language, namely reliability [2]. This means that the resulting code is less idiomatic, and thus less modern, which holds true for comparisons both between versions and crates. This, however, only works in a single direction, as a low amount of unsafe code does not necessarily lead to modern code. This means that, while a high value in this metric likely means lower modernity, it cannot be used on its own.

While the plot of the percentage of `async`-marked functions in Fig. 5 looks similar to the unsafe plot, for the purposes of measuring code modernity it is far less useful. While this plot perfectly picks out crates in the particular niche of server code, with both asynchronous runtimes and web frameworks ranking highly, this ends up not being relevant to our target measure of code modernity. This is also reflected by the trend line having a small coefficient, and staying flat.

The most promising metric is the amount of Clippy warnings per expression, as seen in Fig. 6. This figure shows a very strong trend down to zero warnings as time goes on. For both comparisons between versions of the same crate and comparisons between different crates, this proves to be a highly effective metric to measure code modernity, as after stabilizing, the metric almost universally goes down over time. This makes sense, as it is common practice to minimize the number of Clippy warnings in a code base. Since the analysis uses the latest version of Clippy at the time for all versions, as you go back in time, the amount of new warnings, that previously didn't exist, increases.

In conclusion, not a single metric analysed is a reliable measure for code modernity on its own, however a combination of all of them, except `async`-marked function percentage, can give us a good idea of the code modernity of a code base. The measure is most effective for comparison between versions of the same code base, and while it can give an idea of modernity in comparison to other code bases, this should be researched further.

9 CONCLUSION AND FUTURE WORK

This paper has explored the viability of different metrics for the purpose of measuring code modernity in the Rust language. We defined code modernity as a measure that represents the point in time that the code would've been written when using the contemporary features and idioms of that time. We explored the community's opinion and identified 5 additional metrics that indicate idiomatic Rust code, namely edition, MSRV, unsafe usage, async usage and amount of Clippy warnings. Using this last metric, we determined that existing static analysis tools can be a very useful tool to aid in measuring code modernity. Finally, using the Ruvolution tool we have built [11], we have come to the conclusion that the version signature, edition, reported MSRV, the percentage of unsafe expressions and the amount of Clippy warnings per expression can all be used together to form a code modernity measure.

This paper forms a basis that proves that other metrics besides measuring versions can be viable to measure code modernity in Rust. More research can be done in this area to discover which combination of metrics produce the best results, and how to combine the metrics to create a single measure. The version signature metric in particular could be improved, as there is still a lot potential left untouched, such as version-specific AST nodes, similar to the methods used by van den Brink et al. [36], or larger analysis involving more code bases. Additionally, the methods presented in this research did not provide a strong basis for comparing modernity between different code bases.

The scope of this research is kept small on purpose, but as mentioned in section 4.3, this results in the dataset possibly not being reflective of all Rust code as a whole. There is room here to conduct more research on code modernity analysis using bigger datasets to find more universal metrics that work across all code bases.

The inclusion of static analysis tools proved very useful, which potentially opens the door for a universal code modernity tool across languages, as such tools exist for the majority of widely used languages. As the topic of code modernity continues to garner more interest, and better tools are developed utilizing more metrics and producing better measures, they could eventually be integrated into existing code quality analysis pipelines and contribute to more modern code being written across many projects and languages.

REFERENCES

[1] 2024. Rust Community. https://www.rust-lang.org/community Accessed on 2024-01-14.

[2] 2024. Rust Programming Langauge. https://www.rust-lang.org/ Accessed on 2024-01-14.

[3] 2024. The Rust Programming Language Blog. https://blog.rust-lang.org/ Accessed on 2024-01-14.

[4] 2024. The Rust Programming Language Forum. https://users.rust-lang.org/ Accessed on 2024-01-14.

[5] 2024. Rust Subreddit. https://www.reddit.com/r/rust/ Accessed on 2024-01-14.

[6] Chris Admiraal. 2023. Calculating the modernity of popular python projects. https://essay.utwente.nl/94375/ Publisher: University of Twente.

[7] Weihang Lo Alex Crichton, Eric Huss and contributors. 2024. The Manifest Format. In *Cargo Documentation*. https://doc.rust-lang.org/cargo/reference/manifest.html Accessed on 2024-01-14.

[8] Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Nov. 2014), 472–483. https://doi.org/10.1145/2635868.2635901 Conference Name: SIGSOFT/FSE'14: 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering ISBN: 9781450330565 Place: Hong Kong China Publisher: ACM.

[9] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 1–27. https://doi.org/10.1145/3428204

[10] axon q. 2018. Unsound uses of Unsafe in API · Issue #289 · actix/actix-web. https://github.com/actix/actix-web/issues/289 Accessed on 2023-11-21.

[11] Chris Bleeker. 2024. Ruvolution. https://github.com/cb-p/ruvolution 2024-01-19.

[12] Cambridge Dictionary. 2023. modern (adj.). Cambridge University Press & Assessment. https://dictionary.cambridge.org/dictionary/english/modern Accessed on 2023-12-11.

[13] Cambridge Dictionary. 2023. modernity (noun). Cambridge University Press & Assessment. https://dictionary.cambridge.org/dictionary/english/modernity Accessed on 2023-12-11.

[14] Seema Dewangan, Rajwant Singh Rao, Sripriya Roy Chowdhuri, and Manjari Gupta. 2023. Severity Classification of Code Smells Using Machine-Learning Methods. *SN Computer Science* 4 (2023), 1–20. https://api.semanticscholar.org/CorpusID:260291616

[15] Aria Fallah. 2018. actix-web has removed all unsound use of unsafe in its codebase. It's down to less than 15 occurences of unsafe from 100+. www.reddit.com/r/rust/comments/8wlkbe/actixweb_has_removed_all_unsound_use_of_unsafe_in/ Accessed on 2023-11-21.

[16] Aamir Farooq and Vadim Zaytsev. 2021. There is more than one way to zen your Python. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2021)*. Association for Computing Machinery, New York, NY, USA, 68–82. https://doi.org/10.1145/3486608.3486909

[17] Izzet Berke Guducu. 2022. Weighted Abstract Syntax Trees for Program Comprehension in Java. https://essay.utwente.nl/91735/ Publisher: University of Twente.

[18] Matthias Kaak. 2022. Why shouldn't I use the current rustc version? https://users.rust-lang.org/t/why-shouldnt-i-use-the-current-rustc-version/79345 Accessed on 2024-01-14.

[19] Morten Kristensen. 2023. Vermin. https://github.com/netromdk/vermin Accessed on 2023-11-24.

[20] Jorge Leitao. 2021. Should I bump to 2021 edition? www.reddit.com/r/rust/comments/qmph74/should_i_bump_to_2021_edition/ Accessed on 2024-01-14.

[21] Kornel Lesiński. 2024. State of the Rust/Cargo crates ecosystem. https://lib.rs/stats Accessed on 2024-01-14.

[22] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. 2021. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2183–2196. https://doi.org/10.1145/3460120.3484541

[23] Oxford English Dictionary. 2023. modern (adj.). Oxford University Press. https://doi.org/10.1093/OED/6310816251

[24] Oxford English Dictionary. 2023. modernity (noun). Oxford University Press. https://doi.org/10.1093/OED/4837544359

[25] Manish Goregaokar Philip Krones, Oli Scherer and contributors. 2016. Clippy. https://github.com/rust-lang/rust-clippy Accessed on 2023-11-24.

[26] Manish Goregaokar Philip Krones, Oli Scherer and contributors. 2024. Usage. In *Clippy Documentation*. https://doc.rust-lang.org/stable/clippy/usage.html Accessed on 2024-01-14.

[27] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 763–779. https://doi.org/10.1145/3385412.3386036

[28] Joseph Ryan. 2020. 2363-rustdoc_json. In *The Rust RFC Book*. https://rust-lang.github.io/rfcs/2963-rustdoc-json.html Accessed on 2024-01-24.

[29] Rana Sandouka and Hamoud Aljamaan. 2023. Python code smells detection using conventional machine learning models. *PeerJ Computer Science* 9 (2023). https://api.semanticscholar.org/CorpusID:258985567

[30] Robbie Shilliam. 2010. Modernity and Modernization. In *Oxford Research Encyclopedia of International Studies*. https://doi.org/10.1093/acrefore/9780190846626.013.56

[31] Marco Leni simonsan and contributors. 2023. *Rust Design Patterns*. https://rust-unofficial.github.io/patterns/ Accessed on 2023-11-14.

[32] Jonathan L. Steve Klabnik, mdinger and contributors. 2023. *Rust By Example*. https://doc.rust-lang.org/rust-by-example/index.html Accessed on 2023-11-24.

[33] The Rust Release Team. 2022. Announcing Rust 1.63.0. https://blog.rust-lang.org/2022/08/11/Rust-1.63.0.html Accessed on 2024-01-14.

[34] The Rust Survey Team. 2022. Rust Survey 2021 Summary. https://github.com/rust-lang/surveys/blob/main/surveys/2021-annual-survey/questions.md Accessed on 2024-01-14.

[35] David Tolnay. 2016. Cargo Expand. https://github.com/dtolnay/cargo-expand Accessed on 2024-01-24.

[36] Wouter Van den Brink, Marcus Gerhold, and Vadim Zaytsev. 2022. Deriving Modernity Signatures for PHP Systems with Static Analysis. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 181–185. https://doi.org/10.1109/SCAM55253.2022.00027

[37] Oliver Weiler. 2021. Should I use Clippy in all of my projects? www.reddit.com/r/rust/comments/qvu1iy/should_i_use_clippy_in_all_of_my_projects/ Accessed on 2024-01-14.

[38] withoutboats. 2018. 2394-async_await. In *The Rust RFC Book*. https://rust-lang.github.io/rfcs/2394-async_await.html Accessed on 2024-01-14.

[39] Cristian Zubcu. 2023. Effect of Normalization Techniques on Modernity Signatures in Source Code Analysis. https://essay.utwente.nl/96034/ Publisher: University of Twente.