Final Report

Aedan Kearns – aedank@umich.edu

Overview

I selected Clippy (https://github.com/rust-lang/rust-clippy) as the project I contributed to. It is not a project for social good. Clippy is a linter for the Rust programming language, a static analysis tool used by developers writing Rust to check for errors and ensure that code is performant and idiomatic. This is opposed to the Rust compiler itself, which largely only cares about correctness and syntax errors. Clippy does this by using the already existing Rust compiler code – made available as it is also open source – along with its own additional internal utilities to parse users' code and detect potential issues. It also provides helpful suggestions on how to resolve the issues it spots by providing fixes using the users own code, which in some cases can be automatically applied. Clippy is included when installing the Rust toolchain, and can be run manually by the developer or used in automated testing by, for example, GitHub to assure code quality.

Project Details

Unlike other core parts of the Rust compiler infrastructure, Clippy isn't maintained by any formal organization, and is instead worked on entirely by volunteers. The project is hosted on GitHub, where all collaborative development takes place, and they also have a forum on Zulip where developers can communicate with each other. The project uses a standard fork-and-pull model, where contributors fork the repository, make desired changes in a feature branch, and then submit a pull request in the original repository to incorporate their changes.

The project's CONTRIBUTING file and the developer guide go further into the process and its requirements: Firstly, testing is an important part of the development process, and all new lints should have an accompanying test file which assures that the lint functions properly. Additionally, Clippy has a host of other tests, including so-called "dogfood" tests, where Clippy runs on its own code, ensuring that it follows its own suggestions. The Clippy developer tools also includes lintcheck, a tool which runs Clippy on a selection of real-world Rust libraries, (called crates), which developers are encouraged to check to make sure that there aren't any false positives. Clippy also requires that the code be formatted properly, and that the standard Rust naming conventions are followed, which can be fixed automatically by the developer tools. Developers are encouraged to run all these checks locally before submitting their PR.

Once the PR is submitted, GitHub automatically runs the above tests on the submitted changes, to ensure quality and also that the changes don't require a merge commit to apply. A reviewer is automatically assigned, and they provide feedback for the submitter to rectify until they deem the PR fit to add. Clippy, like other Rust compiler projects, uses GitHub labels and an automated bot to facilitate this exchange, run tests, and ensure the changes can be added without a merge, automatically notifying relevant people during the process be it the PR author or the reviewer(s).

Tasks

I was able to meet the report requirements by completing the first two of the issues I initially selected in the preliminary report:

- Lint for .clone().into_boxed_str() (https://github.com/rust-lang/rust-clippy/issues/15951)
- Lint suggestion: indirect_boxing (https://github.com/rust-lang/rust-clippy/issues/15373)

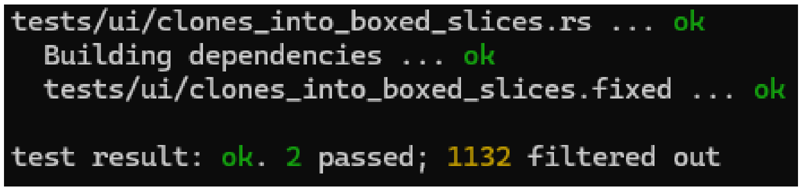
Both issues were making a similar suggestion, so implementing them both ended up being more of a single task. The requests were for a new lint to be added which detects clone-like functions followed by one of several into_boxed_… methods and suggests to replace them with Box::from(), as the former function creates an unnecessary temporary object, whereas the latter does not and is therefore more performant as well as more concise. After creating the new lint and testing it, I created a PR with my changes which would close both of the issues if accepted, and after receiving feedback, fixed the issues and re-uploaded my changes. As of writing the PR is awaiting the new changes to be reviewed. The PR can be found here: https://github.com/rust-lang/rust-clippy/pull/16095.

QA

My overall QA strategy was test-driven development in a spiral model, where I would progressively iterate between receiving test results or feedback and improving my tests or lint accordingly. I chose this approach as I wanted to ensure that I had a concrete understanding and real examples of all of my requirements prior to implementing the lint, so I could be confident throughout development of the correctness of my code. Before starting, I made sure on Zulip that creating a single PR covering both issues was appropriate: https://rust-lang.zulipchat.com/#narrow/channel/257328-clippy/topic/.E2.9C.94.20Clarification.20for.20issue.20.2315951. I also asked for help with the lint's name and category, as both are mentioned in the developer guide as something reviewers look at. After I received confirmation, I started by creating a test which specifically corresponded to my lint, being sure to add both positive and negative cases, and then implemented the lint itself in accordance with the test. This test assures that the lint triggers only when it should, and that the suggestions compile correctly when applied. Determining when a lint should trigger is an important part of adding a new lint. According to the developer guide, unless the lint is in the pedantic category (which mine is not), the lint should never generate false-positives, and instead should err on the side of false-negatives if need be. This is because a warning that is too annoying will often end up ignored instead of being resolved, and Clippy wants to avoid that as much as possible. I opted not to use coverage or mutation testing while developing my test, as Clippy didn't already have support for them, and regardless I found that doing test-driven development resulted in a high quality test from the start. Single test results are shown in image 1.

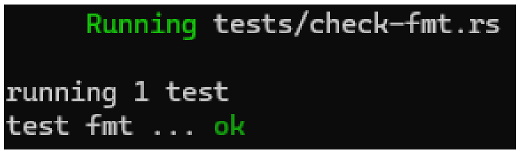
Once I was satisfied with my singular test and my implementation passed it, I then ran the entire test suite and lintcheck. The Clippy test suite runs the other lints on my code to assure its quality, and vice versa (i.e. runs my lint on the entire Clippy codebase) to assure the rest of the codebase follows the new lint. It also checks for formatting and proper naming conventions. Lintcheck, as described in project details, allows for additional manual false-positive checking on real-world code. Images 2 and 3 show relevant test suite results, and image 4 shows lintcheck results.

I figured that repeated communication with the maintainers would create overhead in implementing changes, so I decided testing locally first as much as possible would be the most efficient use of time. After I had done that, I created a PR and then implemented the feedback I got from the reviewers. My implementation, test cases, and communication with the reviewers can be viewed through the link to the PR in the previous section.



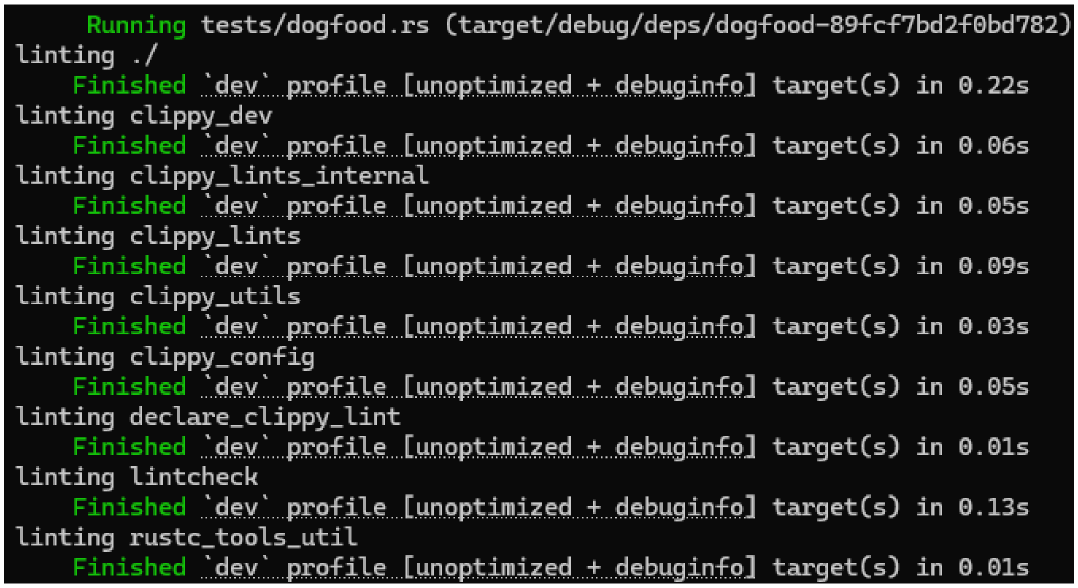
```
tests/ui/clones_into_boxed_slices.rs ... ok
  Building dependencies ... ok
  tests/ui/clones_into_boxed_slices.fixed ... ok

test result: ok. 2 passed; 1132 filtered out
```

1. Single test results



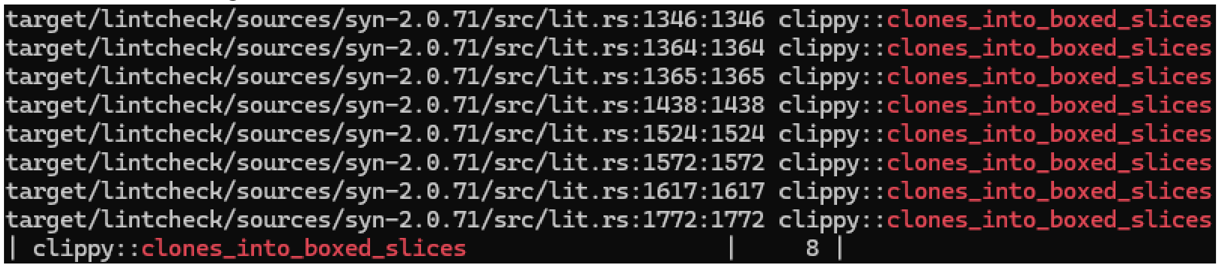
```
     Running tests/check-fmt.rs

running 1 test
test fmt ... ok
```

2. Formatting is correct



```
     Running tests/dogfood.rs (target/debug/deps/dogfood-89fcf7bd2f0bd782)
linting ./
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.22s
linting clippy_dev
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.06s
linting clippy_lints_internal
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.05s
linting clippy_lints
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.09s
linting clippy_utils
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.03s
linting clippy_config
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.05s
linting declare_clippy_lint
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.01s
linting lintcheck
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.13s
linting rustc_tools_util
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.01s
```

3. No errors in dogfood tests



```
target/lintcheck/sources/syn-2.0.71/src/lit.rs:1346:1346 clippy::clones_into_boxed_slices
target/lintcheck/sources/syn-2.0.71/src/lit.rs:1364:1364 clippy::clones_into_boxed_slices
target/lintcheck/sources/syn-2.0.71/src/lit.rs:1365:1365 clippy::clones_into_boxed_slices
target/lintcheck/sources/syn-2.0.71/src/lit.rs:1438:1438 clippy::clones_into_boxed_slices
target/lintcheck/sources/syn-2.0.71/src/lit.rs:1524:1524 clippy::clones_into_boxed_slices
target/lintcheck/sources/syn-2.0.71/src/lit.rs:1572:1572 clippy::clones_into_boxed_slices
target/lintcheck/sources/syn-2.0.71/src/lit.rs:1617:1617 clippy::clones_into_boxed_slices
target/lintcheck/sources/syn-2.0.71/src/lit.rs:1772:1772 clippy::clones_into_boxed_slices
| clippy::clones_into_boxed_slices                       |     8 |
```

4. 8 instances of clones_into_boxed_slices found in the syn crate from a selection of 25 crates

Plan Updates

Working on just the first two issues ended up becoming significantly more time-consuming than I had planned in my initial report, leading to me meeting the final report requirements without having to work on the other three issues listed in the initial plan. Almost all of the sub-task time estimates given in the initial report ended up being significantly less than the actual time I spent. A comparison can be seen in the table below:

Sub-Task	**Estimate (hours)**	**Actual (hours)**
Read contributor documentation	2	3
Development setup	0.5	1
Contact maintainers about first two issues	0.5	0.5
Code familiarization	2	1
Implement unit test	0.5	4
Implement lint	0.5	4
Internal testing and improvements	1	5
External feedback and improvements		4
Developer tools	-	1.5
Total	**7**	**24**

All of the sub-tasks that involved code synthesis took significantly longer than planned. I largely attribute this discrepancy to three factors: The first is that I have no prior experience contributing to a large project like this, meaning I had nothing to base my estimates off of, and the second is my unfamiliarity with the Clippy codebase in particular, making working in it more time consuming. The third factor is that I underestimated how complex the lint I chose to add was, as the lint had to detect and handle several different cases slightly differently to each other with some having additional separate edge cases of their own. The one outlier in the table is the code familiarization sub-task, which I defined as time spent reading code directly and using bottom-up comprehension to understand its purpose, as opposed to reading documentation to understand what code is doing (which I classified under read contributor documentation). Because Clippy is quite well documented, I was able to avoid using bottom-up comprehension most of the time, so I ended up only spending one hour instead of the estimated two.

There were also a few changes in how I tracked and categorized the time I spent compared to my initial plan. In my initial plan, internal testing and addressing feedback were both listed in a combined sub-task, but as previously stated in the QA Strategy section, I ended up doing significant internal testing on my own before making a PR, so I figured it best to split them into their own sub-tasks. Additionally, the developer tools sub-task wasn't in the initial plan at all, which consisted of learning how to use the various tools needed to contribute to a large collaborative project, like learning about git rebasing and how to create and update a PR through the GitHub website.

Experiences

Working in a Large, Unfamiliar Codebase

This was my first experience contributing to a large codebase that I wasn't familiar with, and I was surprised by how different it felt compared to working on school or personal projects. I was expecting there to be more reading and code comprehension than I'd previously experienced, as was discussed in lecture. But one thing I hadn't expected was how working on a project like this affected my decision making, which ended up being more difficult and time-consuming. In previous projects, I was largely just concerned with how to do a given task efficiently, but contributing to a large project like this reframed what I had to consider while making design choices. It was no longer just about "Will this code work?" and "Is this code performant enough?", but also:

- How does this code fit into the larger framework?
- What helper functions exist and where should I use them?
- Is this code readable? Does it need more documentation?
- How do I implement things idiomatically compared to the rest of the codebase?
- What do the maintainers expect to be checked in a lint and test case?

Considering those questions in addition to making sure the code was functional (which is no easy task in itself) was a huge part of why code generation took significantly longer than I had initially estimated. Every time I went to implement something, I had significantly more things to consider, leading to an overhead on all code synthesis. It illustrated to me the difficulty of trying to plan the creation of software. Software engineers are often implementing novel things, and the answers to these questions will be different depending on the requirements of each individual project and what is being implemented. Additionally, the answers to these questions can change over time, which can result in an unpredictable need for a refactor.

Developer Tools and Documentation

One aspect of development that I found surprisingly pleasant was the amount of documentation and automated developer tools that existed for contributors to Clippy. There's a host of documentation specifically for contributors, explaining things like how to set up the project locally, how to implement specific parts of a lint, and the tests and checks recommended before creating a PR. The large amount of documentation was one of the reasons I chose to work on this project in the first place, and I certainly felt its help while developing. As shown in my time tracking, I was able to avoid a lot of bottom-up code comprehension, which is more difficult, and instead rely mostly on top-down comprehension from what was documented.

The documentation wasn't one hundred percent perfect though. One issue that I ran into was that certain kinds of lints in Clippy can be organized into subdirectories which contain additional helper methods for those kinds of lints; the concept of which didn't seem to be very well documented. This led to me not realizing that my lint could've initially been a part of the methods directory, and I ended up re-implementing some of the logic that already existed there. It was only after I submitted my PR and the reviewers pointed it out that I was able to refactor my lint to reside in the methods directory.

That said, the amount of helpful developer tools was the thing that I found the most surprising compared to the documentation. The source code for these tools is included in the repository, and any contributor working on Clippy can compile and run them with a single command to do a variety of tasks automatically. For example, they can create a template lint with a given name for the developer to fill in (dev new_lint), deprecate an old lint and remove related code (dev deprecate), run various different parts of the test suite (dev dogfood; uitest), and automatically generate a test oracle based on a given test case (uibless). Lintcheck, the tool discussed previously, is another one of these. I found these tools to be immensely helpful while developing, and made it much easier as someone new to the project to contribute and assure quality.

Test-driven Development

This was the first time I tried test-driven development, and I found it to be highly beneficial in assuring code quality. It separated *what* the implementation would do from *how* it would do it, allowing me to focus on just one at a time, reducing cognitive load. The test code acted as an unambiguous realization of the requirements, and having them written out allowed me to much more easily spot issues and validate the requirements themselves. Subsequently, having the test case be so detailed, (it ended up having nearly as many lines of code as my actual lint), made it easier to create the lint, as just making it pass the test would act as adequate verification, assuring the correctness of the software with respect to the requirements.

Static Program Analysis

Clippy is a static analysis tool, and working on it gave me a greater appreciation of how difficult static program analysis can be. My lint had to detect two function calls in succession, which while sounding simple, ended up having many difficulties and edge cases. One example is ensuring each function the lint needs to detect is actually the correct function. One of the pieces of feedback I addressed was that just checking the name of a function is not enough, as a programmer could create a function with the same name but with different behaviour, where the lint shouldn't apply. To fix this, I ended up accessing information from Rust's type system, to ensure that each function is called on its correct respective type. Although, this ended up being complex to implement, as each function requires a different type, and the names of the various types sometimes have to be checked differently.

Rust features macros, which allow for programmatic compile-time code generation. Clippy runs on code after macros have been expanded, meaning code that was generated by the macros will be scanned by Clippy. Macro-generated code can often, while being logically correct, have issues with being idiomatic Rust. This could lead Clippy to trigger lints for macro-generated code, but sometimes Rust coders don't have control over what that code is. Because of this, many lints, including mine, choose to simply ignore macro expansions as not to generate false positives beyond a programmers' control, even though sometimes actually it is something that the developer could fix.

An additional thing that I found surprisingly difficult was dealing with all of the ways expressions can be written, and the automatic type conversions of these expressions, especially in relation to creating suggestions. There are many different ways to write logically equivalent code, and suggestions for how to fix linted code need to take that into account. I had multiple instances where I realized that a coder, however unlikely, could code something very strange that would break my lint's suggestion. I ended up either accounting for this in my lint, or choosing err on the side of caution and not generate a lint for that piece of code. Rust also performs automatic dereferencing in method calls, but doesn't for function arguments, so I had to account for this in my suggestions as well. This further demonstrates, along with the difficulties with macros, a common theme among automated tools I've noticed during this semester: that they aren't (and likely can't) be perfect, and that one shouldn't entirely rely on them to assure program quality.

Interaction with Community

One interaction that I had with the community was my post on Zulip, where I asked for clarification on a few things before beginning work on my lint. Specifically, I asked if it would be appropriate to create a single lint combining the two issues and multiple different into_boxed… functions together, what category the lint should belong to, if the lint's suggested name in one of the issues would still be correct with the added scope, and if I could create a PR that referenced more than one GitHub issue. I got replies from two people, one community member and one maintainer, and I felt both to be very helpful, as both went beyond giving a simple answer and helped illuminate some aspects of contributing to Clippy that I hadn't known before. For example, Kevin, a community member, explained (among other things) that claiming multiple issues is fine, but not to worry too much about it, as giving an accurate PR description is more important. Another example is from Jason, the maintainer, where they explained that the term “clones” in the community could refer to any clone-like operation, not just the function literally named “clone”, so the name was still appropriate despite the expansion of scope. In both examples, they could’ve just said “yes that’s fine” or something similar, but they both went into more detail about their reasoning as to why, which I found very helpful in better understanding the community.

Recommendations

If I was a professional software engineer creating a new large project from scratch, I’d replicate many of the things Clippy does and some of my own choices, but also change a few things. I would factor in how difficult and time-consuming code generation can be in a large project, and would be sure to keep that in mind while estimating time required. The large amount of documentation and helpful developer tools are something that I would definitely want to add, while also making sure to add documentation about the overall codebase structure and where things fit into it, as I felt Clippy was slightly lacking in that area. Overall, though, I felt that the documentation and developer tools helped me create changes more quickly, and made me more confident in the quality of my code, which I definitely would like to have. I also found test-driven development to be immensely beneficial, so I would either require or at least heavily encourage its use in my own hypothetical project. Clippy doesn’t use any test quality metrics like code coverage or mutation testing, but I think adding them could be beneficial, and there exist some easy to set up options for Rust. I found that Clippy’s use of a forum outside of just GitHub to also be beneficial, as it provides a way for developers to get help and discuss things in more casual context, and where the entire community can chip in more easily. Overall, I found contributing to Clippy to be a fun and engaging process, and one that was invaluable in learning how to be a software engineer, and I hope to contribute to many more projects (including Clippy) in the future!

Advice for Future Students

As is stated in many places on the course website, start early! Also, be sure to thoroughly go through and understand the readings, because it’s highly relevant to anyone wishing to be a software engineer, and the tests can have questions about them.

Also, I would be fine with future students seeing my materials.