Semantic Querying of Rust Code

Bachelor Thesis Project Description

Nicolas Winkler

Supervised by Prof. Dr. Peter Müller, Vytautas Astrauskas, Federico Poli

Department of Computer Science

ETH Zürich

Zürich, Switzerland

August 15, 2018

1 Introduction

As a relatively young language with a very fast-growing user base, Rust aims to provide an alternative to languages like C or C++, without baiting programmers to make mistakes but rather enforcing safety through restrictions in its type system. Let us take a look at two such rules that are enforced by Rust.

In Rust, it is not possible to create a reference that lives longer than the value it points to. This is implemented by making the so called *lifetime* of a reference part of its type and prohibiting the creation of references pointing to a value that does not exist long enough to match the reference's lifetime. This prevents classic C++ mistakes like returing a reference to a locally scoped value.

```
std::string& get_text()
{
    std::string text =
        "my text";
    return text;
}
```

Listing 1: valid C++ code

```
fn get_text() -> &'static String
{
    let text =
        String::from("my text");
    return &text;
}
```

Listing 2: Invalid Rust equivalent

Although modern C++ compilers will emit a warning when compiling the obviously erroneous code displayed in Listing 1, sometimes the creation of references that outlive the value they reference is more hidden. By making the lifetime of references part of

their type, and prohibiting the creation of references that live longer than their referenced values, this problem can be entirely prevented.

As a second example, array accesses in Rust are bound checked. It is infamously known that C and C++ abstain from doing any bound checks for array access, among other reasons also in order to not lose out on performance. While Rust mostly tries to provide its language features without runtime overhead, in this case Rust takes the sacrifice of checking array access by default and therefore producing safer and better debuggable code. While unchecked array access in combination with incautious programming techniques can lead to possible buffer overflows and exploitable code, checking array bounds prevents a good part of security flaws often encountered in C and C++ code.

There are a few more similar features in Rust for preventing common coding errors which are not presented here, mostly implemented as restrictions in the type system which try to force a good memory management onto the programmer. Although these restrictions are meant to make Rust a safe language, sometimes they are too harsh. In such cases, the programmer needs to circumvent part of these rules by wrapping certain statements of their code inside *unsafe* blocks.

Namely, operations that are only allowed inside an unsafe block or function are [1]:

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait

Let us consider an example usage of the `unsafe` keyword in Listing 3. We want to use a global variable to keep count of the number of events that happened. Modifying and accessing a mutable global variable is forbidden in *safe* Rust, we therefore need to wrap the violating operations in an unsafe block.

```
1  static mut EVENT_COUNTER: i32 = 0;
2
3  fn event_happened() {
4      unsafe { EVENT_COUNTER += 1; }
5      // process event
6  }
7
8  fn main() {
9      // wait for event
10     event_happened();
11     println!("{} events happened!", unsafe { EVENT_COUNTER });
12 }
```

Listing 3: Example usage of global mutable variable

We use the `unsafe` keyword on lines 4 and 11 to access the static mutable event counter variable. By doing this, we promise the compiler that we as a programmer take the precautions to ensure that we will not encounter a data race.

2 Motivation

Rust is still an evolving language and features are regularly added to or removed from the language. Also the Rust community has not yet agreed upon what should be allowed inside unsafe blocks. For removing old features or enforcing new rules for unsafe blocks, it is crucial to analyze existing Rust code and see how often certain coding patterns are used (and how much code would break by changing or removing a feature).

However, manually searching through the whole Rust codebase for certain use patterns is simply not doable, so a tool to automate this task should be developed. One might think of simply downloading the code and running a text-based search tool such as `grep` over it, or maybe use a search engine like found on `https://searchcode.com/`. However, all currently available search tools either only allow text-based search or are not able to parse Rust code.

To get an idea of what kind of queries should be answered, let us take a look at some examples (formulated as English sentences).

- Find all functions that return an `i32` and take no arguments.
- Compute the percentage of unsafe blocks that access a global mutable variable.
- Find all unsafe blocks that call C functions.
- Count the number of unsafe blocks that are possibly executed when calling a certain function.
- Find most important unsafe blocks based on how much code depends on them.

A comparable project, that works for many languages would be SemmleQl [2]. But since it is unclear if it can be extended to support queries of our desired scale and since it currently does not support Rust, it is not suitable for use in this project.

3 Possible Solutions and Ideas

3.1 Query Language

One important part of developing a tool for semantic querying, is to find a suitable query language. It must be expressive enough to allow a user to formulate complex queries, yet it needs to be possible to run given queries efficiently. Since querying the entire Rust codebase (or at least the publicly available part) means querying several gigabytes of Rust code, this need for efficiency becomes even clearer.

Let us take a look at a few options for a query language:

- Basic pattern matching for Rust expressions is used in Rust macros. They can basically be thought of pattern matching and substitution on the abstract syntax tree. However, macros are bound to operate locally on the code, which means for example when searching for function calls, it is not possible to filter by some property of the called function.
- In a blog post [3], Niko Matsakis proposed to use Datalog as a query language for such a tool. Datalog, as a very expressive query language, would certainly be a good choice for this project. As a downside it should be noted that sometimes it can be tedious to formulate very simple queries in Datalog.
- One option would be to use an SQL-like query language. Using the `RECURSIVE` keyword, even complex queries requiring fixpoint iterations can be expressed. However writing raw SQL queries can be very tedious and confusing, so a more clearly laid out wrapping language would be needed.
- Since we basically need to run queries over the abstract syntax tree of Rust code, query languages for XML databases like XQuery are also a feasible possibility. Choosing this option would also require us to check how scalable and performant existing implementations are.
- As a further alternative, it would be possible to give a tree grammar for queries, which can be run over the abstract syntax trees of Rust crates to match certain nodes. This approach would also require to design a suitable query syntax and implement an interpreter for it.

3.2 Implementation

Regardless of the querying method used, to run a query over all public Rust code, the code needs to be downloaded and indexed properly. Rust projects are usually organized in *crates*. A crate is built using Rust's build system and package manager *cargo*, which also manages all dependencies to other crates. The default package repository for cargo, `https://crates.io`, contains at the time of writing 16'873 crates. To maximize the impact of our tool, we want to be able to query all of these crates.

To query multiple gigabytes of code for a certain pattern, it is essential to store information about the code in a way that allows efficient execution of queries. Also it is beneficial to use some framework that can parallelize well over large data sets. One option to run queries would be to encode queries into a language understood by a data analysis framework like Datafrog [4] or Differential Dataflow [5].

4 Core Goals

4.1 Collect Questions

To get an idea of what questions should be possible to answer using the tool and to make the tool fit the Rust community's needs, we would like to collect questions that have come up in discussions, especially in those regarding the unsafe code usage. The tool shall then be built with the goal of delivering answers to these questions.

4.2 Query Language

A suitable query language has to be chosen (or invented, if none suitable exists already) to express the collected questions in the previous goal. For this, both expressivity and performance requirements of this project have to be taken into account.

4.3 Compiler Plugin

To extract all relevant information into an easily queriable format, a plugin for the Rust compiler needs to be created. It shall use the interface that the Rust compiler exposes to IDEs to extract the internal representation of the Rust code. The plugin should then output the gathered information in a queriable and storable format.

4.4 Storage

To store the downloaded and indexed crates, a suitable storage solution has to be chosen. Since an inefficient storage method can quickly become the bottleneck for the search tool, performance aspects have to be taken into account.

4.5 Import Crates

As a last implementation goal, all crates from `https://crates.io/` need to be downloaded, indexed and stored using the previously determined method. This should happen in reasonable time. One must for example pay attention that crates are only downloaded and indexed once, even if they appear many times as dependencies of other crates.

4.6 Evaluation

Evaluate the effort needed with the new system to answer the questions collected earlier. Attention should be brought to the following points:

- Are the yielded results correct?

- Can the collected questions be expressed in the query language and be answered to a satisfying degree?
- How long does a reasonably complex query take to run? Is the tool suitable to answer questions that come up spontaneously during a meeting?
- How complex is the formulation of queries?

5 Extension Goals

5.1 Web-Frontend

To allow the developed tool to be used by the wider Rust community, a web frontend should be developed where queries can be entered and submitted. The query should then be compiled as needed and run in real time. Since the charm of such a frontend is gone when queries take too long to execute, the tool should deliver an answer for most queries in under 5 seconds. This poses a problem when using a larger framework like Differential Dataflow, since compilation usually takes several minutes.

5.2 Incremental Update of Code Database

Another nice-to-have feature would be a technique to redownload and reindex certain packages individually without recreating the whole database. This way, when new versions of a package are published, the search tool could be updated in relatively short time. Depending on how this database will be structured, this can either be a rather easy task or require a bit of sophistication.

5.3 Result Visualisation

Certain types of queries have easily visualizable results. For example, one could query the number of unsafe-blocks that only contain one single function call and compare it to the number of unsafe-blocks used in other ways. To help grasp those numbers, displaying them in a pie chart would be a good option.

While determining, which type of chart is most suitable for a given query is probably best left to the user, the tool should derive what charts can be generated from the query result and provide the user with a selection of suitable charts.

5.4 Compiler Extension

Extend the Rust compiler with an interface that exposes the search function to compiler consumers for example IDEs. This would allow IDEs to implement an advanced search function which can be of use in big projects and across different crates.

6 Schedule

Collect Questions	1 Week
Query Language	2 Weeks
Compiler Plugin	2 Weeks
Database	4 Weeks
Import Crates	2 Weeks
Evaluation	2 Weeks
Extension Goals	2 Weeks
Thesis writing	4 Weeks

References

[1] S. Klabnik, C. Nichols, and Contributions from the Rust Community. (2017) The Rust Programming Language. Accessed on 2018/07/09. [Online]. Available: https://doc.rust-lang.org/book/second-edition/ch19-01-unsafe-rust.html#unsafe-superpowers

[2] (2018) Semmleql. Accessed on 2018/07/13. [Online]. Available: https://semmle.com/ql

[3] N. Matsakis. (2017) Project idea: datalog output from rustc. Accessed on 2018/06/07. [Online]. Available: https://internals.rust-lang.org/t/project-idea-datalog-output-from-rustc/4805

[4] F. McSherry. Datafrog. [Online]. Available: https://github.com/rust-lang-nursery/datafrog

[5] ——. Differential Dataflow. [Online]. Available: https://github.com/frankmcsherry/differential-dataflow