Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semantic Querying of Rust Code

Bachelor Thesis

Nicolas Winkler

January 06, 2018

Advisors: Prof. Dr. Peter Müller, Vytautas Astrauskas, Federico Poli

Department of Computer Science, ETH Zürich

Abstract

To further improve a rapidly evolving programming language such as Rust, it is necessary to analyze how certain features are used. In order to gain insights into how Rust is used, we created a tool which allows queries expressed in a datalog-like query language to be run over large amounts of Rust code. Using this tool, we were able to run certain queries over thousands of Rust projects and get answers in a matter of seconds. It is a good foundation to answer some basic questions and sets a solid base for extensions when searching for certain specific features.

Contents

Contents **iii**

1 Introduction **1**

2 Motivating Queries **3**
2.1 Simple Queries 3
2.2 Recursive Queries 3
2.3 Thief Functions 4

3 The Rust Compiler and Its Ecosystem **5**
3.1 Cargo 5
3.2 Rustc 5

4 Tool Design **7**
4.1 Query Language 7
4.2 Extractor 9
4.3 Linker 10
4.4 Engine 10

5 Implementation and Evaluation **13**
5.1 Working Queries 14
5.2 Performance Measurements 16

6 Conclusions and Future Work **19**
6.1 Possible Improvements 19
6.1.1 Join Ordering 19
6.1.2 Web Frontend 20

Bibliography **21**

Chapter 1

Introduction

Rust is a relatively new systems programming language with growing popularity. It aims to be an alternative for the prevalent C and C++ with a focus on a safe typesystem that ensures memory- and thread-safety.

To still allow very low-level programming, Rust introduced the `unsafe` keyword to denote blocks of code, or whole functions, that may bypass some of the rules enforced by the compiler. Code inside an unsafe-block or inside an unsafe function is essentially allowed to do three things that are otherwise forbidden [6]:

- *Access or update a static mutable variable.*
- *Dereference a raw pointer.*
- *Call unsafe functions.*

By making use of unsafe code, one still needs to uphold the invariants of the typesystem in order to ensure the memory- and thread-safety of Rust code. In other words, when such an error occurs, for example when an access to an invalid memory address is made, it reduces the searching for bugs to these unsafe parts of the code.

What exactly unsafe code is allowed to do as well as what circumstances it can take for granted is part of an active discussion going on[1]. For finding out common use cases that already exist, a tool to analyze the use of certain language features is needed. While some of these analyses, for example measuring the usage of a certain keyword, can be done easily by running a text-based search tool over the code and counting matches, some require more sophistication.

[1] https://github.com/rust-rfcs/unsafe-code-guidelines/

For many programming languages there exist tools, for example Semmle QL [3], that allow for semantic querying. We could not find a tool for Rust that suits our needs.

To answer these types of queries, we developed a prototype for a tool that runs queries expressed in a datalog-like query language on the semantic structure of Rust code. It consists of a wrapper around the Rust compiler that produces a database containing the queriable information. As a second part, we developed a query interpreter that allows queries to be compiled and then run over the database. We will introduce some possible questions we want to answer in Chapter 2, then, after a quick introduction to the Rust ecosystem, we will introduce the design of the tools as well as the choice of query language in Chapter 4. Chapter 5 presents the prototype that was developed and some performance measurements.

Chapter 2

Motivating Queries

To get an idea of what queries the developed tool should be able to answer we want to pose some questions about Rust code in English with the goal in mind to be able to express them as a query and query arbitrary Rust code with it.

Important to note is that there should be a way to query for semantic properties of the code. In contrast to just searching the code with a regex tool, one should for example be able to search for functions that call themselves recursively.

2.1 Simple Queries

There should be a way to "apply a filter" to some elements in the language. For example:

- Find all functions that return an `i32` and take no arguments.
- Find all `if`-blocks that have a single boolean variable as condition.

2.2 Recursive Queries

We also want to be able to compute the transitive closure of some relation between language objects. As an intuitive example, if we maintain some popular library, before removing a deprecated function, we may want to find out what consequences the removal of this function would have. For this we would want to find all functions that can cause the invocation of the deprecated function (i.e. those that would be affected if it were removed). We need to find all functions that may directly call a deprecated function, but also the ones that call a function, which then may call a deprecated function and so forth.

2.3 Thief Functions

In a Letter released on August 8th, 2018 "Detecting Unsafe Raw Pointer Dereferencing Behavior in Rust" [7], Zhijian Huang, Yong Jun Wang, and Jing Liu describe how they searched some crates for what they call *thief functions*. Thief functions are functions that fulfill the following three criteria:

1. *The return value of the function is a mutable reference or data containing mutable references as member fields.*
2. *The input arguments of the function contain no mutable references.*
3. *The function is not declared with unsafe*

These functions are somehow cheating the Rust typesystem and are a possible cause of errors (e.g. they might create multiple mutable references to the same value).

It should also be possible to express a query that filters all functions by these conditions.

Chapter 3

The Rust Compiler and Its Ecosystem

3.1 Cargo

A big advantage of Rust over C and C++ is that it comes with a package manager. Each Rust project is organized into packages, so-called "Crates". Cargo acts as package manager and build system for these packages at the same time. With a single invocation of `cargo build`, we can let Cargo download and compile all necessary dependencies, compile them all and finally build the desired package. An index of published crates can be found on https://crates.io/. At the time of writing, it contains over 20'000 packages that can all be easily downloaded and built with Cargo.

This lies in a heavy contrast to ecosystems like the ones of C and C++, which are not as focused on open source and use many different build tools. This makes it a lot harder to track dependencies between codebases and practically impossible to have a language-global dependency management.

3.2 Rustc

The Rust compiler is called `rustc`. It is itself written in Rust and uses LLVM [5] as a backend.

A standard compilation is carried out in several steps. First, the Rust code is parsed into an AST (abstract syntax tree) representation. The AST is a direct representation of the source code. Each statement, function, control flow block etc. is represented as a node in a big tree. For each crate compiled, one such tree is constructed.

From the AST representation the compiler then generates the so-called HIR (short for high-level intermediate representation). HIR is basically an improved AST, where for example some function call expressions already contain a link to the function they're calling. On the other hand, some informa-

tion is also erased, so are for-loops reduced to infinite loops that contain a conditional `break`-statement.

This HIR is then again lowered into the so called MIR (mid-level IR). During this lowering step, all the type checking is performed. On the MIR, various optimizations can then be carried out. Also the borrow-checker is run.

From MIR, the LLVM-IR is generated on which again optimization passes are run and finally the output binary of the compiler is produced.

For extracting syntactic information about the code, we use the combination of these three representation levels. While HIR and especially AST are more direct representations of the Rust code, MIR contains more type information. Also all static function calls are resolved, a feature that we are using heavily in this project.

Chapter 4

Tool Design

4.1 Query Language

As a query language we experimented with two options of which we then settled on the second one:

- SQL-like syntax
- Datalog-like syntax

SQL provides a simple syntax to express filters and joins over a dataset. A query to search for functions having the same return type and that are calling each other could look like this.

```
select f, g
from f: Function, g: Function
where f.calls(g) && f.return_type() == g.return_type()
```

To extend the SQL-like syntax to support expressing queries that need a fixpoint iteration, we introduced the `UNION`-keyword along with the `Self`-relation, which refers to the relation currently being defined, to express that allows expressing a query that should be run on its own output until a fixpoint is reached.

To search for all functions that possibly call an unsafe function (possibly through other functions in between) we could work with something like this:

```
select f
from f: Function
where f.is_unsafe()
    union
select f
from f: Function, s: Self
```

```
where f.calls(s)
```

To write more complex queries and structure them clearly, it would also be helpful to have named queries. We could rewrite the above query as follows:

```
let IsUnsafe =
    select f
    from f: Function
    where f.is_unsafe();

let CallsUnsafe =
    select f
    from f: IsUnsafe
        union
    select f,
    from f: Function, s: Self
    where f.calls(s);

return CallsUnsafe;
```

As SQL is a widely adopted and well-known query language, this query language would be very easy to get used to for many people. With the extension for denoting recursion, it also allows for very complex queries to be expressed.

When adopting aggregator functions from standard SQL, it would for example also allow querying the number of functions that are called from more other functions than they call functions themselves.

As a downside of the SQL-like approach, it should be mentioned that, while it would allow for very sophisticated queries when implemented like described above, it would need a rather complex query interpreter, for example it would need to enforce a non-trivial type system of the query language.

For the second option we were considering, the one that we finally implemented exposes some predefined relations as datalog facts and lets us define rules to deduce more advanced relations.

The first query, finding all functions that call each other and have the same return type, we write:

```
calls_indirect(F, G) :- calls(F, G).
calls_indirect(F, G) :- calls(F, H), calls_indirect(H, G).

pair(F, G) :- calls_indirect(F, G), return_type(F, T),
                                    return_type(G, T).
```

The rule for pair describes how to deduce our desired relation from a set of predefined facts. We first define the relation of all functions that possibly call each other by starting with all direct calls and then recursively defining the transitive closure of this relation. With the defined relation we can then apply two joins to filter only these tuples that represent two functions returning the same type.

Note that this query language very easily maps to the library functions that datafrog exposes. It is possible to create an interpreter for this language in reasonable time. Also, because it is less complex, query performance will be more predictable. These are the main two reasons why we finally chose the datalog approach.

We wanted to create a tool to query rust code for our described queries. As queries should yield a result quickly we deemed it unfavorable to run queries directly on Rust code, as parsing and semantic analysis would take a lot of time. A suitable design would first iterate over the Rust code to be queried and create an index on which the queries can then be run. Since Cargo provides a nice modular compilation system, we can create these indices on a Crate basis. After we generated an index for each Crate, we want to be able to run queries on all these indices together.

With these thoughts in mind, we structured the tool into the following three parts:

- The **Extractor** is a wrapper around `rustc` which reads out the needed data from the internal data structures of the compiler and stores it into a file per crate.
- The **Linker** then reads all the files generated by the extractor and stores them into a larger database.
- The **Engine** opens the database and waits for input queries which are then processed.

4.2 Extractor

The extractor is implemented as a wrapper around `rustc` that implements a visitor on the HIR datastructure.

For each element that the visitor encounters, it determines whether this element needs to be extracted. If yes, all necessary information about it (e.g. in the case of a function this would be what arguments it takes, what type it returns, what other functions it calls) is stored in a data structure which is itself then part of a larger Crate-`struct`. This data structure is then serialized to a file using the Serde [4] library.

4.3 Linker

The linker deserializes all files created by the extractor and creates one large list of functions, structs, types, modules etc. Each function is assigned a unique global 64-bit index and is stored at this index in the global function list. The same is done for all structs and types etc.

As a next step, all calls of functions to other functions are resolved and stored as tuples of two integers in a new relation. Again the same is done for all other extracted data that can have references across crates.

The merged list is then stored again into one file.

4.4 Engine

The Engine is responsible for compiling and running queries. It parses the input query using a parser generated by LALRPOP [2], a parser generator framework for Rust. The AST of the query is then transformed into a combination of join-, antijoin-, union- and fixpoint operations.

Each datalog rule can contain arbitrarily many clauses separated by a comma, where all clauses but the first one can be negated. The commas basically represent a join if the following clause is not negated, respectively an antijoin, if it is negated. For example the following rule

```
my_rule(A, B) :- first_clause(B), joined_with(B, A), !but_not(A).
```

is transformed into a join on `B` between `first_clause(B)` and `joined_with(B, A)`. The result of this join is then antijoined with `but_not(A)`.

If more than one rule for a relation exists, then a union operation is created that merges all rules together.

If a relation is somewhere used in its own definition, the whole relation is marked as recursively defined. It needs to have at least one base case (i.e. a rule that is not recursively defined).

From this representation, Rust code that uses the Datafrog [1] framework is generated. Datafrog is a very lightweight and fast datalog engine, that exposes methods to perform joins and antijoins. We chose to use it because it exposes a simple interface that fits exactly our needs, it doesn't add any complexity to the system and still performs very well.

The generated Rust code is saved to a temporary file and compiled with `rustc` to produce a dynamic library exposing a function to run the query. This dynamic library is then loaded into the engine and the function to run the query is invoked. We chose this design, because it allows us to keep

the database loaded in memory and run as many queries as desired without reloading the database.

The query language exposes some predefined relations in the query language. An example would be `calls(F, G)` which indicates whether a function `F` calls an other function `G`. Before executing queries, the query engine loads these relations from the database file generated by the linker.

Database Format Internally, all these relations are stored as lists of integer tuples. This allows us to use datafrog to perform very fast join operations on the data. To get back to the name of the function, a lookup in the global function list has to be made.

While this structuring greatly accelerates querying speed, it requires all the tuple data staying in memory, which can use multiple gigabytes of memory when querying large amounts of data.

Chapter 5

Implementation and Evaluation

Due to time constraints, the query tool does not support querying all details of the Rust programming language. It is a rather minimal subset of queries that work. We focused on queries on functions and their attibutes, however the architecture of query language and the engine are not specifically bound to this subset and can be extended to also support queries on individual statements in a function.

The extractor extracts for each function its full path, its argument types, its return type, the paths of all functions it calls and whether it is unsafe or not. For all structs, it stores the type of its fields. For all functions, modules and crates a contains-relationship is extracted.

This exposes the following native relations to the query language:

`calls`, `function`, `in_module`, `modules_in_crates`, `is_unsafe`, `is_struct`, `is_type`, `is_reference_to`, `is_mutable_reference`, `tuple`, `slice`, `argument_types`, `is_struct_type`, `field_types`, `return_type`

The extractor could be extended to extract more features of the code. If desired, one could extract information about all expressions and control flow blocks to analyze more coding patterns.

The linker would analogously need to be extended to support linking the new data together.

The component that would require the least amount of changes in order to add more queriable features to the tool would be the query interpreter. As more extracted information would just appear as more "native" relations that are provided, the parsing and compiling of the queries should not need to change.

5.1 Working Queries

Even though the tool could be extended to support a lot more queries, there are still some interesting queries that work in the prototype.

Type Twins Let us define two functions that have the same return type and where one function directly calls the other one "type twins". A query finding all such type twins can easily be expressed in our query language using only one rule as can be seen in Listing 1.

```
// we declare the new relation that we want to define
// and specify between which types this should be a relation.
decl type_twins(Function, Function);

// definition of the relation `type_twins` as a join of three
// other relations. Notice the free variable `t` used to imply
// the existence of some common return type.
type_twins(f, g) :-
       calls(f, g), return_type(f, t), return_type(g, t).

// define what is done with the results of the query (print all).
!for_each(type_twins, {
    |(f, g)| {
        println!("type twins: {} --> {}",
            data.format_function(f),
            data.format_function(g)
        );
    }
});
```

Listing 1: Query code to find all type twins

Thief Functions We recall our definition of a thief function from the introduction. Basically a function is a thief function, if it returns a value containing a mutable reference, however it does not take anything containing mutable data as an argument.

We can write a query that is filtering all functions by this criterion:

```
// helper relation to indicate which types contain mutable data or references
decl contains_mut(Type);

// which functions return a type containing something mutable
decl returns_mut(Function);
```

```
// which functions have arguments that contain something mutable
decl mutable_arg(Function);

// negation of above relation
decl not_mutable_arg(Function);

// functions not marked as `unsafe`
decl safe_function(Function);

// the relation we are interested in! is a function a thief function?
decl thief(Function);

// used to map functions to their containing crate
decl fn_crate(Function, Crate);

// thief functions with the crate they are located in
decl thief_with_crate(Function, Crate);

contains_mut(t) :- is_mutable_reference(t).
contains_mut(t) :- is_struct_type(t, s), field_types(s, f), contains_mut(f).
contains_mut(t) :- tuple(t, f), contains_mut(f).
contains_mut(t) :- slice(t, f), contains_mut(f).

returns_mut(f) :- return_type(f, t), contains_mut(t).

mutable_arg(f) :- argument_types(f, t), contains_mut(t).
not_mutable_arg(f) :- function(f), !mutable_arg(f).

safe_function(f) :- function(f), !is_unsafe(f).

// we search for all functions that fulfill the mentioned criteria.
thief(f) :- returns_mut(f), not_mutable_arg(f), safe_function(f).

fn_crate(f, c) :- in_module(f, m), modules_in_crates(m, c).

thief_with_crate(f, c) :- thief(f), fn_crate(f, c).

!for_each(thief_with_crate, {
    |(f, c)| {
        println!("function {} in crate {}",
            data.format_function(f),
            data.format_crate(c)
        );
```

```
    }
});
```

Semi-Unsafe Functions To give a simple example of a query that makes use of recursion, we can search for all functions that possibly cause the invocation of an unsafe function.

```
decl possible_unsafe(Function);

possible_unsafe(F) :- is_unsafe(F).
possible_unsafe(F) :- calls(F, G), possible_unsafe(G).

!for_each(possible_unsafe, {
    |(f, )| {
        println!("maybe unsafe: {:?}",
            data.format_function(f));
    }
});
```

5.2 Performance Measurements

To get an idea of the performance of the tool, we compiled some crates using the extractor and measured the running time of the linker and of some queries.

As extracting information from a crate performs a full compilation of it, the performance of the extractor is the bounded by compilation time. It adds negligible overhead to it, therefore we don't provide any measurements for the extractor here.

Since compiling all crates on https://crates.io takes a lot of time and processing power, we chose 9507 arbitrary crates as a test set. These crates combined contain 2'417'817 functions.

We ran some queries on this dataset, to get an idea of the query performance.

Before we can run the queries, the query engine needs to load the database into memory. This loading takes a bit of time (in our case it was about two minutes). However, as the database only needs to be loaded once, and afterwards as many queries as desired can be run, we excluded it from our measurements.

We ran each query 20 times on an Intel® Core™ i7-7600U with 16 GB of LPDDR3 1866 MHz RAM running Debian (4.19.0-1-amd64).

Query	**Average running time [s]**	**Standard Deviation**
Type Twins	0.231	0.00334
Thief Functions	0.673	0.00403
Semi-unsafe Functions	0.128	0.0192

These numbers show us that even for relatively big number of crates, also non-trivial queries execute very fast. We can assume that we would also get reasonable execution times when running queries over all crates published on `https://crates.io`.

Chapter 6

Conclusions and Future Work

While the tool we originally planned to create was supposed to support more queries than it does now, it can do enough to yield some interesting results. The separation between the extractor, linker and query engine proved to be a good design choice that enables fast querying times.

It is a good prototype that can be extended for any project aiming to analyze large amounts of Rust code.

6.1 Possible Improvements

6.1.1 Join Ordering

In our tool, the query compiler naively maps the datalog rules to the underlying joins and antijoins. This puts the burden of ordering the joins on the query writer. Consider the following example showing that sometimes the ordering of joins matters:

```
decl call_each_other(Function, Function);
call_each_other(F, G) :- calls(F, G), calls(G, F).
```

Listing 2: A Query finding functions that both contain a call to the other one

```
decl call_each_other(Function, Function);
call_each_other(F, G) :- function(F), function(G),
                         calls(F, G), calls(G, F).
```

Listing 3: A query equivalent to the one above but much slower and more memory consuming

The query displayed in Listing 2 we can express as a simple join over a binary relation. Since the `calls`-relation only contains pairs of functions, the two additional joins added by Listing 3 don't add any restrictions to the query, i.e. the queries should yield the same result. However when the unnecessary joins are performed from left to right, first a temporary list containing all pairs of functions needs to be generated. For big data sets, this relation can quickly take up terabytes of memory.

A very basic join-ordering query optimizer would already help for such cases and should not be a big problem to implement.

6.1.2 Web Frontend

As queries run relatively fast once the data is extracted, a web frontend could be created where queries can be entered and run over a large number of crates (ideally all crates published on `https://crates.io`).

Bibliography

[1] Datafrog, a lightweight Datalog engine in Rust. https://github.com/lalrpop/lalrpop. Accessed: 2018-12-29.

[2] LALRPOP. https://github.com/lalrpop/lalrpop. Accessed: 2018-12-29.

[3] Semmle QL. https://semmle.com/ql. Accessed: 2017-01-05.

[4] Serde. https://github.com/serde-rs/serde. Accessed: 2018-12-29.

[5] The LLVM Compiler Infrastructure. https://llvm.org/. Accessed: 2018-12-07.

[6] The Rust Programming Language. https://doc.rust-lang.org/1.5.0/book/unsafe.html. Accessed: 2017-01-04.

[7] Zhijian Huang, Yong Jun Wang, and Jing Liu. Detecting unsafe raw pointer dereferencing behavior in rust. *IEICE TRANSACTIONS on Information and Systems*, 101(8):2150–2153, 2018.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Semantic Querying of Rust Code

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Winkler	Nicolas

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 06.01.2019	N. Winkler

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.