# Simple Visualization of Lifetime Errors in Rust

Bachelor Thesis Project Description

David Blaser
Supervised by Prof. Dr. Peter Müller, Federico Poli, Vytautas Astrauskas
Department of Computer Science
ETH Zürich
Zürich, Switzerland

## I. Introduction

Rust [1] is a modern programming language with a focus on memory safety and performance. In this regard it is comparable to C or C++. To ensure memory safety, Rust uses special checks preformed by the so called "borrow checker".

The safety features are mainly enabled by Rust's type system, that is a so called "ownership type system". It assigns an owner to any memory location. When creating references to a memory location, it gets "borrowed". Therefore, these references are considered as a "borrow".

Some restrictions apply to borrows: there may be only one mutable active borrow at a time for each single object. As alternative, if there is no mutable borrow, there may exist multiple immutable borrows. When a new mutable reference (i.e. a borrow) is created, the location it borrowed from becomes blocked. While a location is inactive, it may not be accessed. These properties are checked for all Rust programs at compile time by the so called borrow checker. In order to ensure these constraints, a lifetime is assigned to each variable. Usually the compiler does infer these automatically, but programmers can also provide them explicitly. This allows more fine-grained control. A simple example illustrating borrowing and its limitations is given in Listing 1. The first part of it shows mutable borrowing, whereas the second introduces immutable borrowing.

The newest version of the borrow checker at the time of writing (dubbed NLL) was released last December as part of Rust edition 2018 [2]. In this version, a lifetime is assigned to each variable in a program, representing the timespan in which this variable can be safely used. NLL provide more flexibility to the programmer, and they should make writing a correct program that is accepted by the compiler simpler and more intuitive. This thesis will solely focus on NLL, building upon previous work that was committed as a preceding thesis by Dominik Dietler [3].

Writing Rust programs that adhere to all rules for borrowing and lifetimes can be tricky, and mistakes lead to so called "lifetime errors" that sometimes are hard to understand. Therefore, as main goal of this thesis, we aim to produce visualizations that support programmers in understanding such errors.

## II. Lifetime Errors

The rules described in the previous section are needed to ensure the memory safety guarantees that Rust provides. However, adhering to all restrictions that are imposed by the type system and the borrow checker can be rather complicated. For example, the error messages that are provided by the compiler (in response to lifetime-related errors) are not always easy to understand, especially when multiple lifetimes are involved. Therefore, debugging a lifetime error can be hard, especially for novice Rust programmers.

For example, Listing 2 shows a slightly modified example that was provided in the previous work of Domink Dietler. It contains an error related to borrowing. An experienced Rust programmer will probably be able to easily see the error in this code snippet. However, for a novice programmer, the chain of multiple borrows that are applied to the object which is originally denoted as `x` is possibly hard to understand.

The corresponding error output of the compiler is provided in Listing 3.[1] One can see that an error related to `x` is reported. This is reported to be caused by calling `take(w)`. However, the error message gives no clue on the relation between `x` and `w`. As one can clearly see, the compiler only reports issues on lines 3, 7 and 8. However, these lines are not sufficent to understand what is going on with the object that is created on line 2, and thereby figure out the error in this code. In order to do so, one definitely also needs to consider lines 4 and 5 (plus maybe also line 2), in addition to the lines 3, 7 and 8 that were reported by the compiler.

Clearly, novice programmers would need support that provides help in finding such errors. Unfortunately, the error message that is printed by the compiler does not fully provide this support.

[1] We used the current stable compiler (rustc version 1.33.0). Please note that we (now) consider the Rust edition to be 2018 for this example.

```rust
fn main() {
    let mut x = 42; // x is mutable ...
    x = 12; // ... so we can change it.
    let x1 = &mut x; // we create a reference, x gets mutable borrowed.
    // let x2 = &mut x; // ERROR: cannot mutable borrow x again here.
    // println!("{}", x); // ERROR: cannot access x here, as it is borrowed to x1.
    *x1 = 42; // we can change the object, as the borrow is mutable.
    println!("{}", x1); // however, we can use x1, as it is still active here.

    let y = 42; // y is immutable.
    // y = 43; // ERROR: so we cannot assign to y.
    let y1 = &y; // we can create a (immutable) borrow of it.
    let y2 = &y; // we can also create another (immutable) borrow.
    println!("{}", y); // we can still access (only read) y.
    println!("{}", y1); // we can still access (only read) y1.
    println!("{}", y2); // also y2 is still active here, so we can read it.
}
```

Listing 1: A simple example illustrating borrowing and its constraints. For illustrative purposes, this code does some rather useless things, that cause warnings when it is compiled.

```rust
fn main() {
    let mut x = 4;
    let y = foo(&x);
    let z = bar(&y);
    let w = foobar(&z);
    // ...
    x = 5;
    take(w);
}

fn foo<T>(p: T) -> T { p }

fn bar<T>(p: T) ->T { p }

fn foobar<T>(p: T) ->T { p }

fn take<T>(p: T) { unimplemented!() }
```

Listing 2: Example 3 (slightly modified) from the previous work, containing a non-trivial lifetime error.

## III. Existing Approaches

As mentioned before, this thesis will be based largely on previous work by Dominik Dietler [3]. It already allows acquiring information about a lifetime-related error. This includes all existing lifetimes in a method, all constraints in between of these lifetimes and also the code that these constraints were generated from. More precisely, this information is extracted from the compiler and (especially) from the borrow checker in several (sometimes complex) steps. All in all, the result of this process seems to be of good quality.

However, as of now all of this information is packed into one single graph. As this is quite a lot of information, the resulting graph is rather huge, even for a relatively small input. Furthermore, the resulting graph includes additional internal information that is related to the lifetime error, but is not helpful for understanding the actual error in the input source code.

In order to support this observation, we will shortly analyse the resulting graph for the example 3 from the work of Dominik Dietler. Its code was provided before as Listing 2. Note that its main methods only consists of 6 lines of source, and we only consider the main method of this example. The graph that is emitted by the current tool for it is shown in Figure 1. In total, 23 lifetimes are reported and depicted in the graph, there are 18 constraints between these lifetimes and 6 pairs of lifetimes are considered to be equal. This actually means that these lifetimes are mutually constrained by each other, which is explicitly depicted in the resulting graph. Given such complexity, it is probably legitimate to conclude that this graph is not helpful for debugging example 3.

Also, the current work does not yet include anything that one could consider a useful tool for (novice) Rust programmers: as of now the resulting graph is simply stored as dot file into a certain sub-directly of the current code's project, and the code from the previous work must be run by invoking a rather counter-intuitive make command.

In order to improve the previous work, we could refine the created graph in such a way it only shows information that is actually needed to understand the error. In order to do this, the previous work suggests to elide "anonymous intermediate lifetimes" and "spurious equalities" form the graph [3]. Furthermore, we would like to develop a plug-in for an IDE that allows examining the information right next to the analysed code.

In addition also some more technical, or internal improvements to the current code base would be very welcome. For example, the code (as it is) depends on a

```
error[E0506]: cannot assign to `x` because it is borrowed
--> src/main.rs:7:5
  |
3 |     let y = foo(&x);
  |                 -- borrow of `x` occurs here
...
7 |     x = 5;
  |     ^^^^^ assignment to borrowed `x` occurs here
8 |     take(w);
  |          - borrow later used here

error: aborting due to previous error
```

Listing 3: Compiler error for Example 3. Additional warnings and references to the documentation are omitted.

no longer up-to-date nightly version of the rust compiler toolchain, originating from last summer. Furthermore, the previous work cannot be used without setting this (old) version as default toolchain. Upgrading to a newer toolchain seems to be beneficial, especially as the Rust language (and its compiler infrastructure) are moving fast. As mentioned before, NLL became stable when the Rust edition 2018 was released.

Furthermore, as shortly indicated before, also the build system could probably gain some usability by applying some improvements. Even more steps that could also allow for improvements are pointed out in the next sections.

Another relevant problem is the actual visualization of the lifetimes that exist in a piece of Rust code. Several previous ideas on how lifetimes could be shown to the user in a legible, nice-looking way exist. A proposal by Jeff Walker that specifically focuses on a visualisation inside of an IDE appeared lately [4]. It proposes to depict lifetimes as coloured lines next to the source code. The shape of a line shall give more information about the lifetime and the variable it belongs to. This topic is also extensively discussed on the Rust Internals forum [5].

## IV. Core Goals

In this thesis we aim to build tools that shall support programmers in understanding and fixing errors related to lifetimes and borrowing. To do this, complete and helpful information about lifetimes (that cause errors) should be provided to programmers in a clear, simple and especially understandable fashion. The previous work by Dominik Dietler [3] already allows to extract a lot of useful information about lifetimes form the compiler and the borrow checker. However, the information is displayed in a form that is not yet that helpful for programmers, as mentioned before.

The main goal of this thesis is to extend and improve the previous work, and thereby get a tool that is useful for (novice) Rust programmers to understand and debug errors related to lifetimes.

- **Goal 1 (Familiarize with previous work):**
  Get into the previous work, and thereby apply some first improvements. This e.g. includes moving to a newer Rust compiler version, and improving the build system. Some more (technical) improvements might be necessary.
- **Goal 2 (Collect examples):**
  This can be done in parallel to the first goal. As of now, we have less then ten example code fragments that contain an lifetime error that is caused by the borrow checker. In order to test and evaluate our implementation, we would like to get more such examples.
  We consider several possible ways to do so. We could write some code by ourself, e.g. implement some data structures, and record errors that we come across. In addition, we could take some existing implementation, e.g. data structures from Prusti [6], and try to introduce a single lifetime error. Also an option would be to ask for help from the Rust community, i.e. asking on the Rust Internals forum, as presumably a lot of people already run into such errors. Yet another option would include looking for possible issues on Stack Overflow or GitHub, but most of the errors appearing there are probably related to old versions of the borrow checker. However, some useful examples could possibly be found in the Rust language issue tracker.
  Another option that should be considered is looking at the specification of the borrow checker, and then try to write examples against it, i.e. writing snippets that try to violate (exactly) one of the specified constraints.
- **Goal 3 (Improve the Graph):**
  As extensively described before, the resulting graph is not yet optimal to help programmers in understanding borrow-checker errors. Therefore, this representation should be improved. We believe that this could mainly be achieved by leaving out unnecessary information. This probably includes most of the intermediate variables and internal intermediate lifetimes. Another idea includes to merge lifetimes that are equivalent. In addition, one could also consider to

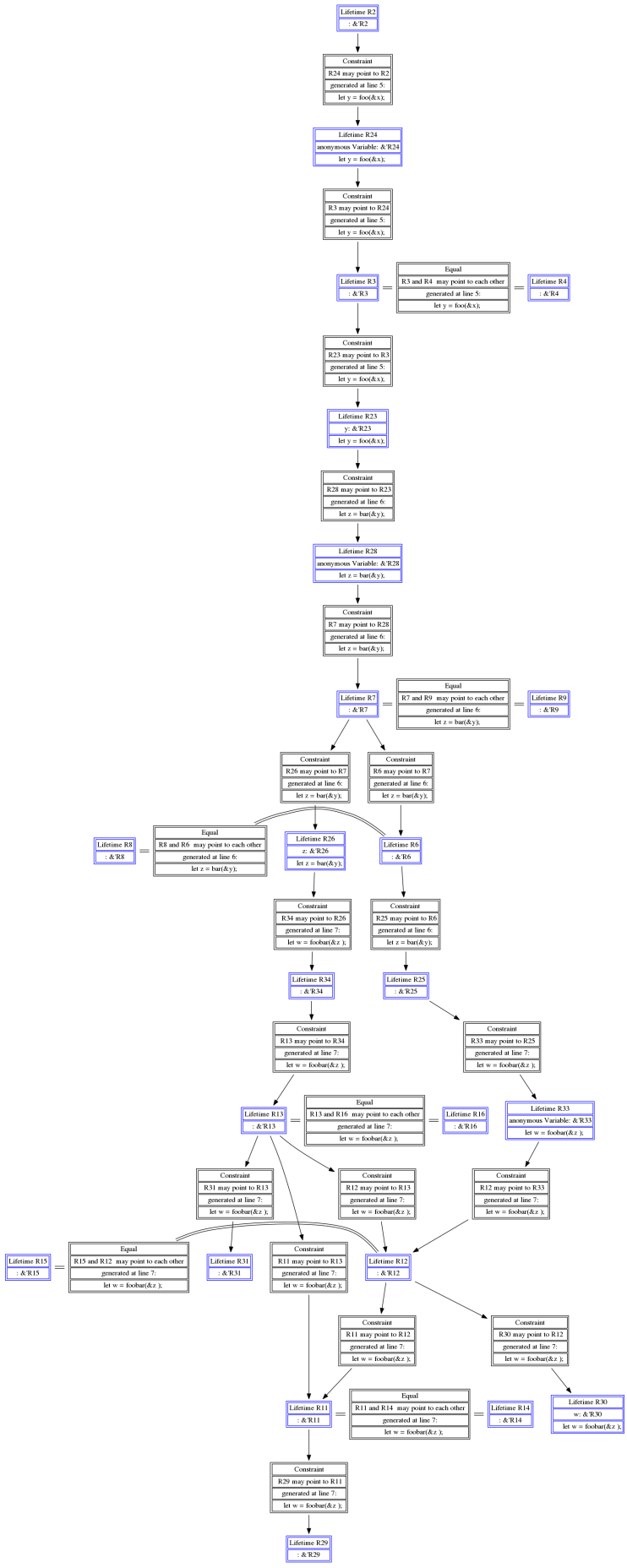


Fig. 1. Resulting graph for example 3, that was given in Listing 2.

only keep the shortest direct path through the entire graph.
As there are different proposals, we will need to do some try-outs and then define which simplifications actually provide improvements, while not loosing any relevant information.

- **Goal 4 (Linking of lifetimes to source lines):**
In the previous work, all lifetimes are linked to the line of source where they are (presumably) introduced. However, this is based on a very simple heuristic. It simply takes the first line (lowest line number) that is part of the lifetime. This works well for the current examples, but it should be tested more extensively. E.g. this was never tested with examples that include loops.
- **Goal 5 (IDE Extension: display graph):**
We want to write an extension for an IDE that uses the information we can acquire with our tool and helps programmers to debug their lifetime-related errors. A first step probably includes showing the resulting graph inside of the IDE.
- **Goal 6 (IDE Extension: link to code):**
As part of this goal, we would like to allow for some way(s) to interact with the graph. Primarily we want to point out the correspondence between the graph nodes and the lines of code. Maybe this could be achieved by highlighting code and graph nodes when they are activated by the programmer, e.g. by a mouse click, a mouse-over or a keyboard shortcut.
- **Goal 7 (Evaluation):**
Once the implementation is complete, we should also evaluate our work. E.g. we must check if the information is now really shown in a way that is easier to understand. Furthermore, we must test if our IDE plug-in (if we succeed in writing it) is also actually helpful for Rust programmers. In order to evaluate the tool, we plan to use cognitive walktrough [7], or a similar method.

## V. Possible Extension Goals

- **Different representations:**
In the main part, we will keep the focus on presenting the lifetime information as a graph. However, other representations would be possible. So it might be worth it to evaluate, and eventually implement these as well. One option would include representing each lifetime on a timeline. Maybe one could also get some inspiration from the proposal by Jeff Walker [4]. Another alternative would include to actually visualize the lifetimes as the stack (and heap) structure they are resulting in. This could be helpful for more experienced programmers, esp. those that are into low level (e.g. C) programming.
- **Handle unsafe blocks:**
Unsafe code is currently not considered as part of this work. However, as it introduces a completely new dimension and also is regularly used in the wild, trying to add support for it in some form might be beneficial.
- **Receive User Feedback:**
As an extension, to do a better evaluation of the IDE plug-in, if we succeeded in implementing it, we could ask Rust programmers to try using the tool. For this, the tool could be advertised to the Rust users community at ETH Zurich or on the Rust Internals forum. The users should be asked to use the tool, either with a sample project or for their productive work, and give feedback about it. Such feedback would probably be helpful for evaluating the plug-in, and also for getting new ideas for future improvements.

## VI. Schedule

We give estimates for the number of weeks that are needed for each core goal and other tasks.

- Goal 1 (Familiarize with previous work): 3 Weeks
- Goal 2 (Get more examples): 1 Week
- Goal 3 (Improve the Graph): 3 Weeks
- Goal 4 (Linking of lifetimes to source lines): 3 Weeks
- Goal 5 (IDE Extension: display graph): 2 Weeks
- Goal 6 (IDE Extension: link to code): 2 Weeks
- Goal 7 (Evaluation): 1 Week
- Extension goals: 1 Week
- Writing final report: 4 Weeks

## References


[1] N. D. Matsakis and F. S. Klock, II, "The Rust Language", in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14. New York, NY, USA: ACM, 2014, pp. 103–104. [Online]. Available: http://doi.acm.org/10.1145/2663171.2663188

[2] (2018) Announcing Rust 1.31 and Rust 2018. Accessed on 2019/03/11. [Online]. Available: https://blog.rust-lang.org/2018/12/06/Rust-1.31-and-rust-2018.html#non-lexical-lifetimes

[3] D. Dietler, "Visualization of Lifetime Constraints in Rust", 2018. [Online]. Available: https://www.ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Dominik_Dietler_BA_report.pdf

[4] J. Walker. (2019) Rust Lifetime Visualization Ideas. Accessed on 2019/03/12. [Online]. Available: https://blog.adamant-lang.org/2019/rust-lifetime-visualization-ideas

[5] Borrow visualizer for the Rust Language Service. Accessed on 2019/03/12. [Online]. Available: https://internals.rust-lang.org/t/borrow-visualizer-for-the-rust-language-service

[6] Prusti. Accessed on 2019/03/12. [Online]. Available: http://www.pm.inf.ethz.ch/research/prusti.html

[7] Task-Centered User Interface Design : 4. Evaluating the Design Without Users. Accessed on 2019/03/12. [Online]. Available: http://hcibib.org/tcuid/chap-4.html#4-1