# Visualization of Reference Lifetimes in Rust

Bachelor Thesis Project Description

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

## I. Introduction

Rust [1] is a modern programming language with a strong focus on safety, concurrency and speed. It is comparable to C in terms of performance. Among other features, its type system is designed to prevent data races, as well as dangling pointers and null dereferences.

One of the novelties of Rust is the distinction between two kind of references: shared `&T` and mutable `&mut T`. Shared references can only be used to read memory they are pointing to, while mutable ones can also be used to mutate the memory they are pointing to. Having two mutable or a mutable and a shared reference to the same memory location at one program point can lead to data races, for example if a thread reads the value of a shared reference while another is mutating the same memory location via a mutable reference. For this reason the type system only allows two situations:

- having one or more shared references (`&T`) to a memory location
- or having exactly one mutable reference (`&mut T`) to it.

To be able to ensure that these two situations are mutually exclusive, the compiler needs to know at which program points a specific reference may be used. For this reason, each reference is annotated with a "lifetime" that informs the compiler for how long a reference can be safely used. In an abstract sense, lifetimes correspond to a set of program points. Lifetimes can be implicit or explicit. In Listing 1, the function foo takes a reference x with an implicit lifetime and the function bar takes x with an explicit lifetime `'a`.

```
1  // implicit
2  fn foo(x: &i32) {
3  }
4
5  // explicit
6  fn bar<'a>(x: &'a i32) {
7  }
```

Listing 1: Implicit and explicit lifetimes

Explicitly writing lifetimes gives more control to the developer, for example, to express that the return value of a function must live for the same lifetime as its arguments:

```
fn x_or_y<'a>(x: &'a str, y: &'a str) -> &'a str { .. }
```

This means that x, y and the return value need to be alive for at least the set of program points corresponding to lifetime `'a`. If x and y need to have different lifetimes, one can use multiple lifetime parameters:

```
fn x_or_y<'a, 'b>(x: &'a str, y: &'b str) -> &'a str { .. }
```

In this example, y is alive for the set of program points denoted by lifetime `'b`, which may or may not be the same as the set corresponding to `'a`. There can be specific constraints between two lifetimes. For example, when there is a reference to a struct with a field that holds another reference, that second reference needs to live at least as long as the one to the struct. In the following example the constraint is implicit and will be inferred by the compiler:

```
fn x_or_y<'a, 'b>(x: &'b Struct<'a>) -> &'b str { .. }
```

For more control, constraints can also be explicitly written with `'a:'b`:

```
fn x_or_y<'a, 'b>(x: &'a str, y: &'b str) -> &'a str where 'a:'b { .. }
```

The `'a:'b` is commonly called an "outlives relationship", in this case lifetime `'a` outlives lifetime `'b`. This means that whenever a reference with lifetime `'b` is alive, so has to be a reference with lifetime `'a`.

There are two different implementations of lifetimes: the original one based on the lexical scope, called lexical lifetimes, and the new one called non-lexical lifetimes.

The lexical lifetime checks are very conservative: they reject programs that would run fine at runtime. Listing 2 is an example of such a program.

```
struct T(i32);

fn main() {
    let x = T(123);

    let y = &x;         // y holds a reference to x until
                        // the end of the main function,
    let z = x;          // so we get an error here
}
```

Listing 2: Error with lexical lifetimes

This can be fixed by manually restricting the lexical lifetime of `y`, as shown in Listing 3. But that is detrimental to the readability of the code and can be challenging to get right. A better solution, seen in Listing 4, is to use the new non-lexical lifetimes, which are precise enough to accept the program in Listing 2. With non-lexical lifetimes, lifetimes are computed to be the minimal set of program points needed to satisfy all corresponding constraints, rather than a lexical scope and hence the name [2]. The advantage of non-lexical lifetimes is that they are more flexible: lexical lifetimes usually last until the end of the containing block, while non-lexical lifetimes aim to be as short as possible and thus are more accurate and may for example only last for a couple of statements or only for one branch of an if-statement. However, this makes them more complicated and harder to understand.

```rust
struct T(i32);

fn main() {
    let x = T(123);

    {
        let y = &x;      // here y is only alive to
    }                    // this curly brace

    let z = x;
}
```

Listing 3: Manually restricting lifetime

```rust
#![feature(nll)]

struct T(i32);

fn main() {
    let x = T(123);

    let y = &x;        // since y is never used in this block after here,
                       // it is not alive anymore here
    let z = x;
}
```

Listing 4: Using non-lexical lifetimes

## II. The Problem

Lifetimes are a great advantage for the programmer, because the compiler can use them to automatically check that references are used only while they are valid. On the other hand, understanding how the compiler performs its checks on lifetimes can be hard for a beginner, especially when the compiler reports errors involving multiple lifetimes.

Listing 5 is a simple program that does not compile because of a lifetime error, which is reported in Listing 6. Even though in this case the error message is clear and simple, a visualization would help to understand what is going on. On real world projects the compiler may produce more complicated, longer error messages. In the appendix (Listings 7 and 8) there are a few examples.

An example visualization are the comments in Listing 5. This is the style used by tutorials and the official documentation [3] to explain lifetimes. Manually writing and maintaining this kind of comments is very error prone and time consuming, especially when editing the code after writing the comments, since one has to be careful not to break the formatting, which would make the visualization useless. Moreover, this visualization does not explain what causes the error: it would be useful to visualize enough information for progammers to understand how to fix it.

For the reasons mentioned above, it would be helpful to have a tool that automatically generates and displays lifetime information near or within the code, for example in a pop-up when hovering over a reference or in a column next to the code. That would help both beginners and advanced developers, and the Rust community already expressed interest in such visualizations [4].

```rust
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let x;                      // -+ x goes into scope
                                //  |
    {                           //  |
        let y = &5;             // ---+ y goes into scope
        let f = Foo { x: y };   // ---+ f goes into scope
        x = &f.x;               //  | | error here
    }                           // ---+ f and y go out of scope
                                //  |
    println!("{}", x);          //  |
}                               // -+ x goes out of scope
```

Listing 5: Simple program with visualization of a lifetime error currently shown in Rust tutorials

```
error[E0597]: `f.x` does not live long enough
  --> src/main.rs:11:14
   |
11 |         x = &f.x;              //  | | error here
   |              ^^^ borrowed value does not live long enough
12 |     }                          // ---+ f and y go out of scope
   |     - `f.x` dropped here while still borrowed
...
15 | }                              // -+ x goes out of scope
   | - borrowed value needs to live until here

error: aborting due to previous error
```

Listing 6: Lifetime error of Listing 5

## III. Core Goals

The goal of this project is to build a tool that will help Rust developers to reason about lifetimes and related compiler errors, by visualizing lifetimes and (optionally) constraints between them.

**Goal 1 (Code Collection/Examination):** Collecting various Rust programs from Stack Overflow, GitHub and similar sources that either compile fine or do not compile due to lifetime errors. Manually examining the collected programs to understand where developers struggle with lifetimes and where and what kind of information will be the most helpful.

**Goal 2 (Choosing Visualization):** Based on information gained in Goal 1, deciding what kind of visualization would be most effective in helping developers to understand lifetime errors, or generally reason about lifetimes. Possibilities could be something similar to visualization in comments, seen in Listing 5, in a column next to the code or a pop-up, similar to what IDEs use to display auto complete options and other information.

**Goal 3 (Compiler Modification):** Modifying the Rust compiler in order to expose the lifetime information that needs to be displayed. One possibility would be to make the data available via the same API used by IDE tools, so that a front end tool can access it. Another option could be to write the information to a file.

**Goal 4 (Tool Development):** Developing a tool, for example an IDE plug-in, that interacts with the modified Rust compiler and visualizes the lifetime information. Another possibility would be to provide the visualization as an extension to the Rust Playground [5], a tool to compile and experiment with Rust code snippets online.

**Goal 5 (Evaluation):** Evaluating the end result. First, checking if the problems identified in Goal 1 are solved by the tool. We are planing to use cognitive walkthrough [6] or a similar method. Second, evaluating response time and stability of the tool.

## IV. Possible Extensions

**Other Visualizations:** A possible extension to the project regards the visualization, not only of lifetimes, but also of other kinds of information, extracted from the compiler. For example, it would be possible to visualize also constraints between lifetimes and highlight what generated them.

**Documentation Generation:** Developing a tool that generates visuals that can be used in Rust tutorials or documentations for explaining Rust borrows and errors related to them. Tools used to write tutorials do not support interactive elements, so we are limited to static elements like code snippets or images. These would be code with comments or images containing highlighted code and lifetime visualizations.

**User Studies:** Another extension could be to conduct user studies to further evaluate and get more feedback on the tool developed in Goal 4. There could be a controlled study with two groups, both would be given a series of programs with lifetime errors to fix, but only one would use the tool. Then it would be compared how many of the problems each group could solve and how fast.

## V. Schedule

- Goal 1 (Code Examination): 1 weeks
- Goal 2 (Choosing Visualization): 2 weeks
- Goal 3 (Compiler Modification): 6 weeks
- Goal 4 (Tool Development): 4 weeks
- Goal 5 (Evaluation): 2 week
- Extensions: 2 weeks
- Writing report: 5 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.* New York, NY, USA: ACM, 2014, http://doi.acm.org/10.1145/2663171.2663188.

[2] N. D. Matsakis, "An alias-based formulation of the borrow checker," April 2018, [accessed 19 Jun. 2018]. [Online]. Available: http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/#fnref:covariant

[3] "The Rust Programming Language," Apr 2016, [accessed 19 Jun. 2018]. [Online]. Available: https://doc.rust-lang.org/1.8.0/book

[4] "Borrow visualizer for the Rust Language Service," Oct 2016, [accessed 24 Jun. 2018]. [Online]. Available: https://internals.rust-lang.org/t/borrow-visualizer-for-the-rust-language-service/4187

[5] J. Goulding, "Rust Playground," Jul 2018, [accessed 18. Jul. 2018]. [Online]. Available: https://github.com/integer32llc/rust-playground

[6] "Task-Centered User Interface Design : 4. Evaluating the Design Without Users," Jul 2018, [accessed 18. Jul. 2018]. [Online]. Available: http://hcibib.org/tcuid/chap-4.html#4-1

```
error[E0597]: borrowed value does not live long enough
  --> src/main.rs:14:5
   |
14 |     Parser { context: &context }.parse()
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ temporary value does not live long enough
15 | }
   | - temporary value only lives until here
   |
note: borrowed value must be valid for the anonymous lifetime #1 defined on the
 ↪  function body at 13:1...
  --> src/main.rs:13:1
   |
13 | / fn parse_context(context: Context) -> Result<(), &str> {
14 | |     Parser { context: &context }.parse()
15 | | }
   | |_^

error[E0597]: `context` does not live long enough
  --> src/main.rs:14:24
   |
14 |     Parser { context: &context }.parse()
   |                       ^^^^^^^ borrowed value does not live long enough
15 | }
   | - borrowed value only lives until here
   |
note: borrowed value must be valid for the anonymous lifetime #1 defined on the
 ↪  function body at 13:1...
  --> src/main.rs:13:1
   |
13 | / fn parse_context(context: Context) -> Result<(), &str> {
14 | |     Parser { context: &context }.parse()
15 | | }
   | |_^
```

Listing 7: Example of a complicated lifetime error

```
error[E0495]: cannot infer an appropriate lifetime for autoref due to conflicting requirements
  --> prusti-viper/src/procedures_table.rs:42:40
   |
42 |         let mut cfg = self.cfg_factory.new_cfg_method(
   |                                        ^^^^^^^^^^^^^^
   |
note: first, the lifetime cannot outlive the anonymous lifetime #1 defined on the method body at 40:5...
  --> prusti-viper/src/procedures_table.rs:40:5
   |
40 | /     pub fn set_used(&mut self, proc_def_id: ProcedureDefId) {
41 | |         let procedure = self.env.get_procedure(proc_def_id);
42 | |         let mut cfg = self.cfg_factory.new_cfg_method(
43 | |             // method name
...  |
135| |         self.procedures.insert(proc_def_id, method);
136| |     }
   | |_____^
note: ...so that reference does not outlive borrowed content
  --> prusti-viper/src/procedures_table.rs:42:23
   |
42 |         let mut cfg = self.cfg_factory.new_cfg_method(
   |                       ^^^^^^^^^^^^^^^^
note: but, the lifetime must be valid for the lifetime 'v as defined on the impl at 22:1...
  --> prusti-viper/src/procedures_table.rs:22:1
   |
22 | impl<'v, P: Procedure> ProceduresTable<'v, P> {
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   = note: ...so that the expression is assignable:
           expected viper::Method<'v>
              found viper::Method<'_>

error: aborting due to previous error
```

Listing 8: Another complicated lifetime of error