Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Visualization of Lifetime Constraints in Rust

Bachelor Thesis

Dietler Dominik

December 19, 2018

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

Department of Computer Science, ETH Zürich

Abstract

Rust is a modern programming language with a strong focus on speed, safety and concurrency. To that end the Rust compiler checks for data-races and similar memory safety issues at compile time using so called lifetimes to statically compute the timespan in which references can be safely used. However, in case of error the compiler messages can be difficult to understand, especially for beginners.

In this thesis, we provide an algorithm that identifies a set of source code lines that complement the error message and are in practice sufficient to reason about the error. We further define a visualization of the constraints between lifetimes that the compiler uses internally to identify the error. We implemented these in a prototype that generates a graph of the constraints and shows the identified source code lines.

We evaluated the prototype over several examples, automatically checking that the visualized constraints are indeed sufficient to generate the error, and manually checking that all the source code lines relevant to the error and not already visualized in the compiler error message are identified.

Contents

Contents **iii**

1 Introduction **1**
1.1 Motivation 1
1.2 Contributions 2

2 Background **3**
2.1 Lifetimes 3
2.2 Lifetime Constraints 5
2.3 Lifetime Errors 6
2.4 Polonius 6

3 Design **9**
3.1 Code Examination 9
3.2 Choosing Visualization 10

4 Implementation **13**
4.1 Information Extraction 13
4.1.1 Extracting Constraints 13
4.1.2 Finding Source 14
4.2 Visualizing 15

5 Evaluation **19**

6 Future Work **21**

A Datalog Rules **23**
A.1 Polonius Rules 23
A.2 Inverted Rules 24

Bibliography **27**

Chapter 1

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. Rust uses a part of the compiler called borrow-checker to check for these kinds of errors at compile time.

1.1 Motivation

The error messages generated by the borrow checker are notoriously difficult to understand for beginners, because they require reasoning on not just the lines reported by the compiler. For example, the error message for the code example in listing 1.1 would point to the lines (a), (c) and (d), but not line (b), which the user should also look at to understand the error.

To easily understand these errors more information is needed, than is given in the error messages. To help with this issue we want to provide a tool that complements these error messages with additional information.

```
1 fn foo(v: i32) {
2     x = &v;   // (a)
3     // ...
4     y = &x;   // (b)
5     // ...
6     v += 1;   // (c)
7     take(y;) // (d)
8 }
```

1.2 Contributions

This thesis addresses the previously mentioned problem by:

1. Providing an algorithm that identifies a set of source code lines that relate the line at which the borrow occurs (a) with the line at which the borrow is later used (d);
2. Defining a visualization of the lifetime constraints that have been used by the borrow checker to identify the lifetime error;
3. Providing a prototype that implements the previous two points.

Chapter 2

Background

2.1 Lifetimes

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`.

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:

```
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

```
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 are two different implementations of lifetimes: the original one based on the lexical scope, called lexical lifetimes, and the new more precise 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.

```
1  struct T(i32);
2
3  fn main() {
4      let x = T(123);
5
6      let y = &x;          // y holds a reference to x until
7                           // the end of the main function,
8      let z = x;           // so we get an error here
9  }
```

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.

```
1  struct T(i32);
2
3  fn main() {
4      let x = T(123);
5
6      {
7          let y = &x;      // here y is only alive to
8      }                    // this curly brace
9
10     let z = x;
11 }
```

Listing 3: Manually restricting lifetime

```
1  #![feature(nll)]
2
3  struct T(i32);
4
5  fn main() {
6      let x = T(123);
7
8      let y = &x;       // since y is never used in this block after here,
9                        // it is not alive anymore here
10     let z = x;
11 }
```

Listing 4: Using non-lexical lifetimes

2.2 Lifetime Constraints

Lifetimes alone are not enough to detect unsafe reference accesses, constraints between the lifetimes are also needed. 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`.

2.3 Lifetime Errors

Compilation errors generated by the borrow checker use the so-called three points error message [3]. For example, consider the simple program in listing 5. This example does not compile because of a lifetime error, which is reported in listing 6.

```
1 fn main() {
2     let mut x = 4;
3     let y = &x;
4     let z = &y;
5     x = 5;
6     take(z);
7 }
```

Listing 5: Simple program with lifetime error

The error message points out the first borrowing location (line 3), the forbidden usage of the borrowed location (line 5) and the later usage of the borrow (line 6). However it does not show why the the first borrowing location and the later usage of the borrow are related, especially when those lines use completely different variables.

```
error[E0506]: cannot assign to `x` because it is borrowed
  --> src/main.rs:9:5
   |
3  |     let y = &x;
   |             -- borrow of `x` occurs here
...
5  |     x = 5;
   |     ^^^^^ assignment to borrowed `x` occurs here
6  |     take(z);
   |          - borrow later used here

error: aborting due to previous error
```

Listing 6: Lifetime error of listing 5

2.4 Polonius

The so called borrow-checker is the part of the compiler which uses lifetimes to perform the checks explained in section 2.1, such as checking for dangling pointers. The borrow-checker works on an intermediate represen-

tation based on a control-flow graph. Polonius [4], the newest version of the borrow-checker, is the component that implements the non-lexical lifetime checks. It uses rules defined in the Datalog language [5] to check for errors. These rules have been implemented using the Datafrog engine [6], a Rust library designed to be a fast engine for executing datalog-like rules. Listing 7 shows one of the rules used in Polonius.

```
loan_live_at(L, P) :-
    region_live_at(R, P),
    requires(R, L, P).
```

Listing 7: Datalog rule used in Polonius

The complete set of Polonius datalog rules can be found in appendix A.1.

Chapter 3

Design

To decide what information exactly we want to display, we collected and examined code examples with lifetime errors. Based on these findings we defined a visualization.

3.1 Code Examination

The first step was to collect code examples with lifetime errors to examine the code and to determine what kind of information would be needed to understand the errors better. Since we focused on non-lexical lifetime errors, which are still an experimental feature and not yet in the stable release of Rust, we did not find many useful examples online. Because of that we constructed our own examples containing various errors related to lifetimes.

To find out what information would help to understand the error we manually examined the programs to see what we need to reason about the error. Then we compared the information we used to reason about the error with the information given by the error message.

We found that the main reason why the errors are so difficult is that the error message does not explain how the lines that are reported interact together to form the lifetime error.

For example, the error message, shown in listing 8, for the code in listing 9 only reports lines 3, 8 and 9, but to understand how lines 3 and 9 are connected we also have to look at lines 5 and 7. However, we can safely ignore lines 4, 6 and 8.

The collected examples can be found in the collected_code directory of the project repository [7].

```
    error[E0506]: cannot assign to `x` because it is borrowed
     --> example3.rs:8:5
      |
    3 |     let y = foo(&x);
      |                 -- borrow of `x` occurs here
    ...
    9 |     x = 5;
      |     ^^^^^ assignment to borrowed `x` occurs here
   10 |     take(w);
      |          - borrow later used here

error: aborting due to previous error
```

Listing 8: Error of listing 9

```
1  fn main() {
2      let mut x = 4;
3      let y = foo(&x);
4      //\ldots
5      let z = bar(&y);
6      //\ldots
7      let w = foobar(&z );
8      //\ldots
9      x = 5;
10     take(w);
11 }
```

Listing 9: Example 3 of the collected Examples

3.2 Choosing Visualization

While choosing the visualization we considered existing projects. The main project was Borrow Visualizer for the Rust Language Service [8], which used the approach of highlighting the state of a selected borrow on the source code. In this thesis, however we focused on lifetime constraints and the source code lines that generate the constraints, because we found that this information is the most helpful for understanding lifetime errors better.

We experimented with different ways to visualize lifetimes and the constraints between them and found that the best way to visualize lifetime constraints would be a directed graph. We defined nodes for the lifetimes and constraints and the edges to show the direction of the constraints. This resulted in a graph where, for two lifetimes 'a and 'b and a constraint 'a:'b, there is a directed edge from node 'a to node 'a:'b, and one from node 'a:'b to node 'b, as shown in figure 3.1.

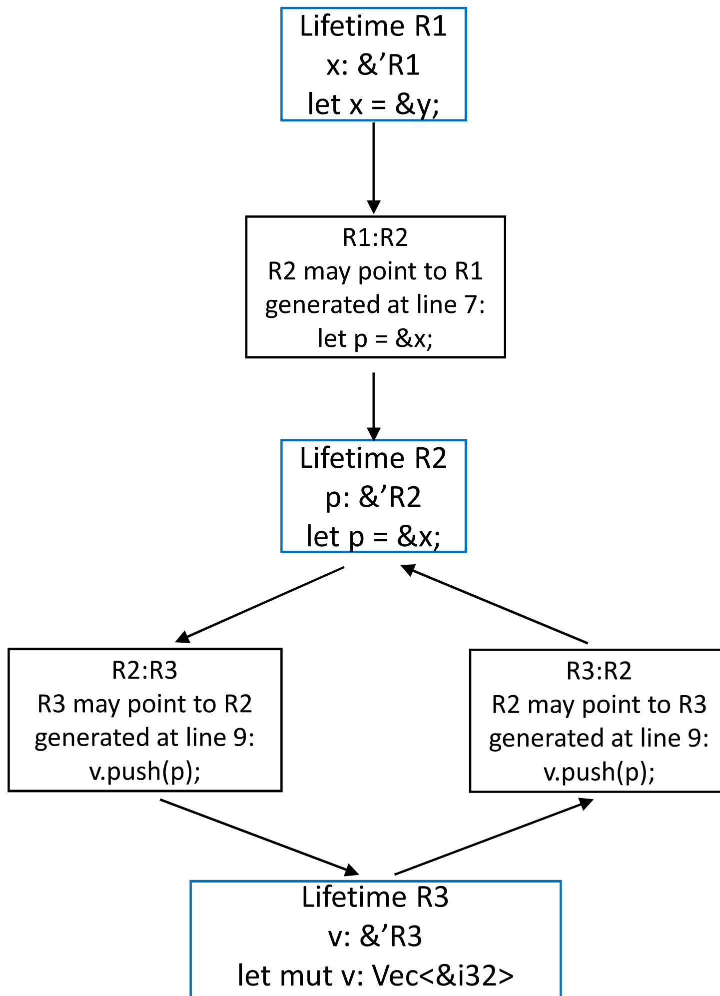


Figure 3.1: Graph with Constraints as Nodes

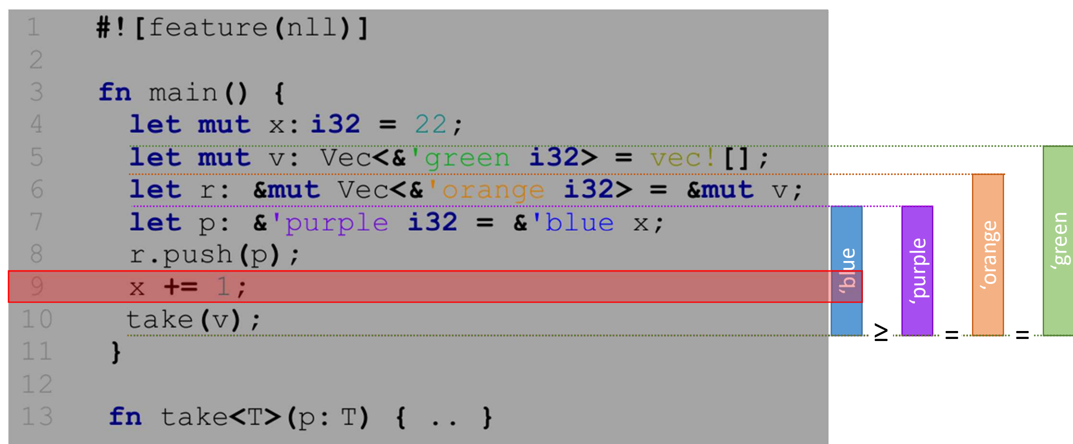


Figure 3.2: One Possible Visualization of Lifetimes

The information we want to display in our graph is

- For Lifetimes:
  - the name of the lifetime,
  - the variable or reference it is associated with,
  - the source code line where it starts
- For Constraints:
  - the constraint
  - the line number where it was generated
  - the source code line where it was generated

Another approach that we tried, before choosing the constraints-based visualization, was to represent each lifetime as a bar next to the code, as seen in Figure 3.2. It soon became clear that this visualization could become quite confusing with increasing size of code and number of involved lifetimes. Displaying a lifetime bar also does not answer the fundamental question, that is why the lifetime has a particular length, and does not end somewhere before. This question is better answered by explicitly stating what are the constraints between lifetimes, and which line caused them. This is why we decided to base our visualization on the lifetime constraints instead of the lifetimes themselves.

Chapter 4

Implementation

We implemented the visualization described in section 3.2 as a compiler plug-in, that extracts information from the compiler and displays it to complement borrow-checker error messages. During the implementation one of the main challenges was to reverse engineer and extract information from the compiler, as described in the following section.

4.1 Information Extraction

The information we want to display was obtained in multiple steps. As presented in the following sections we first extracted the constraints involved in generating the error from the borrow-checker, then we linked those to the source code via compiler dumps.

4.1.1 Extracting Constraints

To extract the constraints we wrote a modified version of the borrow-checker algorithm with additional rules to compute the cause of the lifetime error and ran this over the input and output facts of the original algorithm.

The set of facts on which Polonius operates is quite large even for relatively simple programs and grows with the size and complexity of the program. To get a small subset of facts sufficient to generate the error we inverted the Datalog rules of Polonius, similar to an approach suggested in [2]. To invert the Polonius rule that computes the error, shown in listing 10, for example,

```
error(P) :-
    invalidates(P, L),
    loan_live_at(L, P).
```

Listing 10: Polonius Rule for Error

we generated a new rule for the right-hand-fact "invalidates" and for "loan_live_at", like shown in listing 11.. We called these rules "inverted" because they work backwards: given one error they generate the "invalidates" and "loan_live_at" facts that were needed to generate the error.

```
expl_invalidates(P, L) :-
    expl_error(P),
    invalidates(P, L),
    loan_live_at(L, P).

expl_loan_live_at(L, P) :-
    expl_error(P),
    invalidates(P, L),
    loan_live_at(L, P).
```

Listing 11: Inverted Error Rule

To invert a general rule with n facts on the right-hand side, we construct n new rules, each with an additional fact on the right-hand side for the fact on the left-hand side of the original rule. This is shown in listing 12.

The complete set of the inverted rules can be found in appendix A.2.

4.1.2 Finding Source

After computing the set of lifetime constraints and regions to be visualized, the following step was to link them to an appropriate source code line, such that the user can understand which line caused a particular constraint, or which declaration introduced a particular lifetime.

Our extraction algorithm returns a tuple consisting of the constraint and the program point it was reported at for every constraint computed in section 4.1.1. Because the same constraint is reported at multiple program points, we get many tuples for the same constraint. To keep the graph as simple as possible, we wanted to show every constraint only once. We found that only showing the earliest occurrence of every constraint is a good heuristic to identify the line that generated that constraint. We look up the program point corresponding to that constraint in a compiler internal data structure to find the associated span of source code.

To link the lifetimes involved in the constraint to the source code a detour was necessary. We wrote regular expressions to extract the compiler internal local variables associated with the lifetimes of our constraints from compiler dumps. Then we could look up these local variables in a compiler internal data structure to get the corresponding span of source code at which the variable is defined.

```
A(a_1, a_2, ...) :-
    B(b_1, b_2, ...),
    C(c_1, c_2, ...),
    ...

    \Downarrow

expl_B(b_1, b_2, ...) :-
    expl_A(a_1, a_2, ...),
    B(b_1, b_2, ...),
    C(c_1, c_2, ...),
    ...

expl_C(c_1, c_2, ...) :-
    expl_A(a_1, a_2, ...),
    B(b_1, b_2, ...),
    C(c_1, c_2, ...),
    ...

...
```

Listing 12: Inversion of a general Rule

4.2 Visualizing

We use the DOT language [9], a graph description language, to define the graph and the GraphViz [10] software to display it.

One change to the graph design presented in section 3.2 that we made, was to merge two constraint nodes if one was the inversion of the other, since this means that the two involved lifetimes are equal, which can be expressed as a single constraint as shown in figure 4.1.

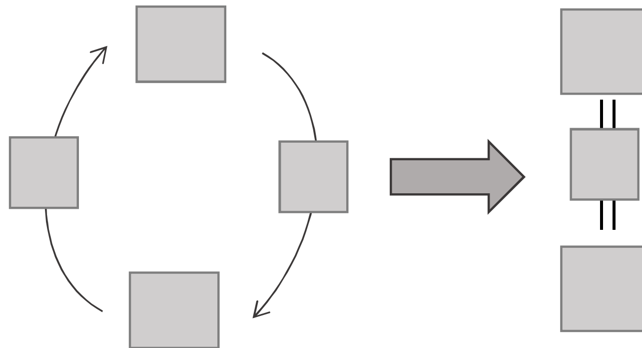


Figure 4.1: Merging Constraint Nodes

This simplification made the graph clearer and more readable, because it reduced the number of nodes and edges by a considerable amount for some errors.

Figure 4.2 shows the graph the prototype generates for the example in section 3.1.

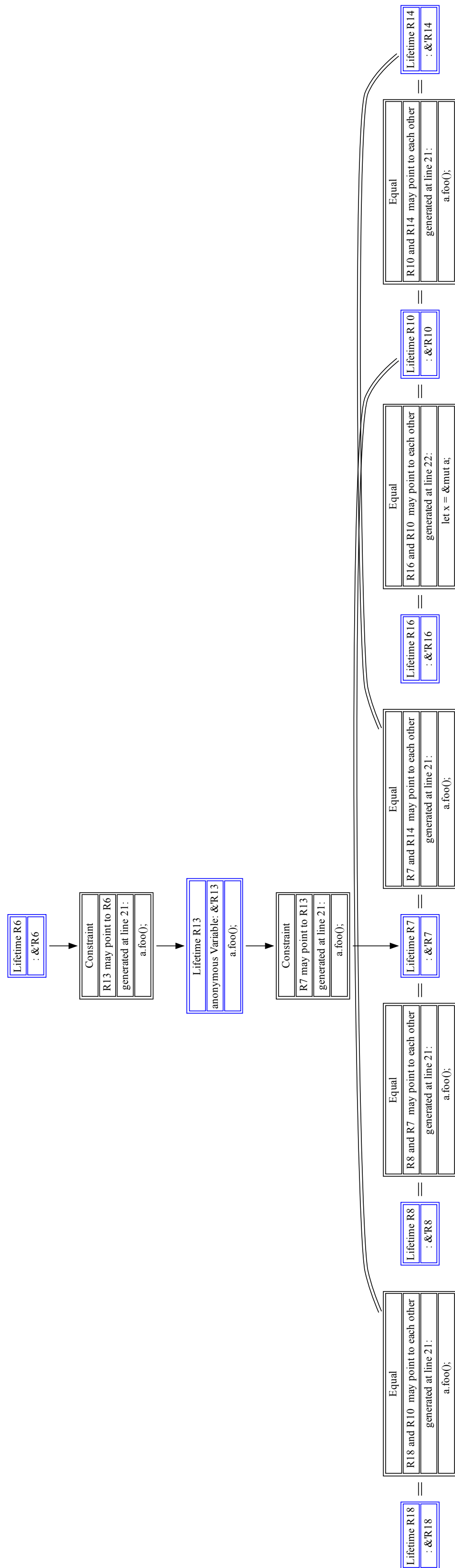


Figure 4.2: Graph produced by prototype

Chapter 5

Evaluation

To evaluate the prototype, we ran an automated test to check that we computed the correct constraints and manually examined if the correct source code lines are displayed.

For the automated test we rerun Polonius with just the subset of constraints identified in section 4.1.1, to check that the set is actually sufficient to generate the error. This test was successful for every evaluated example that contained a borrow-checker error.

To test the identified source code lines we manually examined the code and noted the lines needed to reason about the error. We then checked that the lines displayed by the prototype are indeed a super-set of the ones identified in the previous step.

Given above criteria, table 5.1 shows the results of the evaluation for the examples collected in section 3.1.

example1.rs	$\sim$
example2.rs	✓
example3.rs	✓
example4.rs	✓
example5.rs	✓
example6.rs	✓

Table 5.1: Evaluation Results

In table 5.1, ✓ means that the graph displays exactly the lines we expected. $\sim$ says that all the expected lines are shown, but also some additional ones we did not expect.

Upon further inspection some of the collected examples were determined to

be outside the scope of the thesis because the error they cause is only related to lifetimes and not produced by the borrow-checker.

Chapter 6

Future Work

With this thesis we have developed a prototype that complements borrow-checker error messages with additional information. However there are still possible improvements and extensions that could be further explored. For one, the graph of constraints, produced by the prototype, could possibly be simplified by exploring ways to remove unneeded information like spurious equalities and anonymous intermediate lifetimes. Second, a IDE-plugin could be written, to highlight and display the information provided by our prototype inside the IDE.

Appendix A

Datalog Rules

A.1 Polonius Rules

```
error(P) :-
    invalidates(P, L),
    loan_live_at(L, P).

loan_live_at(L, P) :-
  region_live_at(R, P),
   requires(R, L, P).

 requires(R2, B, P) :-
   requires(R1, B, P),
   subset(R1, R2, P).



requires(R, B, Q) :-
   requires(R, B, P),
   !killed(B, P),
   cfg_edge(P, Q),
   region_live_at(R, Q).


subset(R1, R2, P) :-
   outlives(R1, R2, P).
```

```
subset(R1, R3, P) :-
   subset(R1, R2, P),
   subset(R2, R3, P).
```

```
subset(R1, R2, Q) :-
   subset(R1, R2, P),
   cfg_edge(P, Q),
   region_live_at(R1, Q),
   region_live_at(R2, Q).
```

A.2 Inverted Rules

```
expl_loan_live_at(L, P) :-
   expl_error(P),
   invalidates(P, L),
   loan_live_at(L, P).
```

```
expl_requires(R, L, P) :-
   expl_loan_live_at(L, P)
   region_live_at(R, P),
   requires(R, L, P).
```

```
expl_requires(R1, B, P) :-
   expl_requires(R2, B, P)
   requires(R1, B, P),
   subset(R1, R2, P).
```

```
expl_subset(R1, R2, P) :-
   expl_requires(R2, B, P)
   requires(R1, B, P),
   subset(R1, R2, P).

expl_requires(R, B, P) :-
   expl_requires(R, B, Q),
   requires(R, B, P),
   !killed(B, P),
   cfg_edge(P, Q),
   region_live_at(R, Q).


expl_outlives(R1, R2, P) :-
   expl_subset(R1, R2, P),
   outlives(R1, R2, P).
```

```
expl_subset(R1,R2, P) :-
   expl_subset(R1, R3, P),
   subset(R1, R2, P),
   subset(R2, R3, P).

expl_subset(R2, R3, P) :-
   expl_subset(R1, R3, P),
   subset(R1, R2, P),
   subset(R2, R3, P).


expl_subset(R1, R2, P) :-
   expl_subset(R1, R2, Q),
   subset(R1, R2, P),
   cfg_edge(P, Q),
   region_live_at(R1, Q),
   region_live_at(R2, Q).
```

Bibliography

[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. [Online]. Available: 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] ——, "Three point error form," [accessed 15 Dec. 2018]. [Online]. Available: https://rust-lang.github.io/rfcs/2094-nll.html#leveraging-intuition-framing-errors-in-terms-of-points

[4] ——, "Polonius," May 2018, [accessed 15 Dec. 2018]. [Online]. Available: https://github.com/rust-lang-nursery/polonius

[5] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)," *IEEE Trans. on Knowl. and Data Eng.*, vol. 1, no. 1, pp. 146–166, Mar. 1989. [Online]. Available: https://doi.org/10.1109/69.43410

[6] F. McSherry, "Datafrog," May 2018, [accessed 18 Dec. 2018]. [Online]. Available: https://github.com/frankmcsherry/datafrog

[7] D. Dietler, "Prototype implementation of the tool," Dec 2018, [accessed 19 Dec. 2018]. [Online]. Available: https://gitlab.inf.ethz.ch/vastraus/bsc-ddietler

[8] P. D. Faria, "Borrow visualizer for the rust language service," Oct 2016, [accessed 18 Dec. 2018]. [Online]. Available: https://internals.rust-lang.org/t/borrow-visualizer-for-the-rust-language-service/4187

[9] "Dot language," [accessed 18 Dec. 2018]. [Online]. Available: https://www.graphviz.org/doc/info/lang.html

[10] "Graphviz," [accessed 18 Dec. 2018]. [Online]. Available: https://www.graphviz.org/

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):

Visualization of Lifetime Constraints in Rust

Authored by (in block letters):

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

Name(s):	**First name(s):**
Dietler	Dominik

With my signature I confirm that

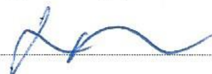
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

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

Place, date

Grüt, 19.12.2018

Signature(s)



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