Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Simple Explanation of Complex Lifetime Errors in Rust

Bachelor Thesis

David Blaser

August 23, 2019

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

Department of Computer Science, ETH Zürich

Abstract

The Rust programming language provides memory safety features by its type system that allows sharing of resources in a controlled and safe way. Violating the rules of this system can lead to lifetime errors. These can be hard to understand, especially for novice Rust programmers. Such errors get complex when more then three lines of source code are involved in the error. In those cases the Rust compiler is not able to provide a complete, understandable explanation.

To tackle this issue we propose Rust Life: A tool that closes the gap that is left by the compiler and creates a simple, step-by-step explanation for many cases of lifetime errors. This is done by analysing information that was extracted from the Rust compiler in various ways, and especially by processing the inputs and outputs of the Datalog-based component that checks lifetimes.

To increase its usability we have created Rust Life Assistant, an IDE extension that allows a simple usage of Rust Life. It is able to display explanations for lifetime error either in the form of a graph or as a text-based guide. The elements of these explanations are graphically linked to their corresponding relevant lines of source code. Our evaluation shows that it can provide complete and helpful explanations for most of the examples that we have collected. We believe that the presented technique could be implemented in the Rust compiler, for the benefit of both beginners and advanced Rust developers.

Contents

Chapter 1

Introduction

Rust [1] is a modern, emergent 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, the location 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 blocked, 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.

This thesis does solely focus on NLL[1], building upon previous work that was committed as a preceding thesis by Dominik Dietler [2].

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. Especially for more complex errors, when the Rust

[1] Short for Non-Lexical Lifetimes. They are explained shortly in section 2.1.

```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.

compiler unfortunately also does not provide a complete and understandable explanation for the source(s) of such errors.

In this thesis we present "Rust Life", a tool that helps programmers to understand lifetime errors and their causes. Thereby it aims to provide support to easily and quickly debug such errors. For this, the tools creates a visualization or a text-based explanation for the error in a given program. These resources are easily accessible through an IDE extension, the so called "Rust Life Assistant". It also enables highlighting of the lines of code that are relevant for understanding the error directly in the editor window.

To get the needed information, Rust Life starts running the Rust compiler on the program that shall be analysed. Then, all needed information, e.g. about the lifetimes of the program, is extracted from the running compiler. This data is then further processed. By some clever tricks and computations information that is needed for an understandable explanation of a lifetime error is acquired. This information is finally visualized and displayed to the user.

We have evaluated the Rust Life Assistant, and thereby also Rust Life on a set of examples that we have collected either over the time of implementing Rust Life or crafted manually. For most of these examples Rust Life run successfully and was mostly able to provide a complete, helpful explanation for the lifetime error in the example. However, some limitations and some small issues of Rust Life were detected. These are discussed as part of the evaluation in chapter 5.

The next chapter will give a short introduction of relevant concepts, it is directly followed by an extensive explanation of the problems imposed by complex lifetime errors. Afterwards, our proposed solution is largely explained in chapter 4, that also gives some rather low-level implementation details. Finally, there is a chapter describing our evaluation and presenting its results, which is followed by a conclusion that lists possible further work.

Chapter 2

Background

2.1 Polonius, the NLL borrow checker

The borrow checker is a component of the Rust compiler. It checks that there are no lifetime errors in a Rust program that shall be compiled. In 2018 Polonius was first introduced in a blog post by Matsakis [3]. Polonius is the borrow checker for "Non-Lexical Lifetimes" (NLL) [4]. For borrow-checking with NLL 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.[1]

An overview of the rules that it applies is given in Appendix A. More precisely, the overview gives the naive Polonius rules.[2] Since the release of Rust edition 2018 in December 2018 NLL, and hence, the Polonius borrow checker, are used by default [4].

Polonius does operate on so-called input facts, generated by the previous compilation steps from the source code of the input program. By applying its rules Polonius will then compute its output facts. These will reveal lifetime errors in the input program.

All of these facts are defined in terms of lifetimes (these are also called regions), of loans (that are also called borrows, especially in newer versions) and in terms of Points in the program's control flow graph. Detailed explanations about these concepts can be found in the original blog post [3] and in the Appendix A.

[1] In comparison to older borrow-checking mechanisms.

[2] As counterpart to the naive rules there are also versions that are optimized for a more efficient computation, these are used by the Rust compiler. (Unless it is instructed to do otherwise.) However, for this thesis we are using the naive rules.

2.2 The Rust compiler

The software that was written for this thesis does interoperate closely with the Rust compiler. However, understanding these internals should not be necessary for understanding this thesis. Still, the interested reader can find more information in the the "Rustc guide" [5] and in the documentation of the current nightly compiler-internal API [6].[3] One detail that shall be pointed out here is "MIR". This is short for "Mid-level Intermediate Representation" [7], and describes an intermediate representation that is used by the Rust compiler. The software described in this thesis mostly operates on the level of MIR.

[3] However, it should be noted that this API is unstable and therefore might change any time. It also is only available in nightly builds of the Rust compiler.

Chapter 3

Lifetime Errors can be hard

3.1 Compiler errors in Rust, or "Is the compiler kidding me?"

An unique feature of Rust are its compilers diverse, ubiquitous error messages. Anyone that ever tried out Rust has probably seen them. In a well-structured way these error messages try to explain to the programmer the reasons for which the compiler is not accepting the program it was asked to compile. In many cases these error messages point out the exact point in the program source that is causing the error, and often it provides a simple hint that allows for fixing some errors within seconds.

It is true that the Rust compiler does emit many errors (especially when facing inexperienced Rust programmers), also reporting about issues that other language's compiler would definitely never complain about. Therefore, the compiler is sometimes perceived as slightly intractable. A friend of the author once described writing Rust code as "A fight against the compiler", when speaking about the process of tuning ones code in order to "convince" compiler of accepting the written program. However, when one starts reasoning about these errors, one might note that the error one was pointed to by the Rust compiler actually had lead to a runtime failure in a different programming language. This does not only apply when comparing to dynamically typed languages (like Python), but some issues would also (potentially) cause runtime failures in some statically typed languages (like C++). E.g. when accidentally writing incorrect C++ one often gets undefined behaviour or a segmentation fault at runtime that do not provide any further information about their cause.

So the Rust compiler is actually saving programmers from executing programs that would (eventually) crash anyway, and often also supports programmers in fixing the bug. Instead, with another language, the program

will start running, until it eventually fails in some way, sometimes not even giving a single hint to the actual error.

However, lifetime errors in Rust are often hard [8][1]. One of the causes for this is probably that lifetime errors do not exist in most other (mainstream) programming languages, so they are a new appearance for many programmers. Also, many programmers might not (yet) know the underlying concept of lifetimes, and it is also not exactly a simple concept. Unfortunately, the claim that lifetime errors are hard seems to also apply for the compiler. In this case one might end up asking oneself: "Is the compiler kidding me?"

As intended, it will report any lifetime error, and it also will provide an explanation for them. For this it will (mostly) use so called "three point error messages" [9][2]. These provided a understandable explanation for simple cases, and it is also acceptable that the compiler does not provide any hints for potential fixes for lifetime errors. (This is probably really impossible in the case of lifetime errors.) Unfortunately, when the errors get more complex, e.g. when more then two variables are involved in causing the error, these error messages get really confusing. In such cases, the compiler error messages will not provide all information that is needed for fully understanding the entire lifetime error. In such cases, one must look at several lines to understand the error. However, the compiler will not point out all of them. In the next section 3.2 this will be explained in more detail by providing an example.

We conclude: Of course, the Rust compiler does not kid programmers, but in some cases it will not provide all relevant information, and hence it is not sufficiently helpful. Programmers are left completely alone. With Rust Life we propose a way to provide some extra help.

3.2 Simple example

This section will illustrate the problem that was pointed out before with a simple, but illustrative example. Its source code is given in Listing 1.[3] Even though this example is simple, it contains a non-trivial lifetime error. An experienced Rust programmer will probably be able to spot and understand the lifetime error in this example quickly.

[1] Actually we intended to quote a following part of this blog post series that was announce to discuss lifetimes. However, this part was not (yet) released at the time of writing.

[2] The concept was first introduced in the NLL RFC, in the section "How We Teach This".

[3] This is the same example that was also given in the proposal for this thesis, and it is basically equivalent to "Example 3" ('example3.rs') from our example collection and is taken from the previous work of Domink Dietler.

```
1  fn main() {
2      let mut x = 4;
3      let y = foo(&x);
4      let z = bar(&y);
5      let w = foobar(&z);
6      // ...
7      x = 5;
8      take(w);
9  }
10
11 fn foo<T>(p: T) -> T { p }
12
13 fn bar<T>(p: T) ->T { p }
14
15 fn foobar<T>(p: T) ->T { p }
16
17 fn take<T>(p: T) { unimplemented!() }
```

Listing 2: Example 3 (slightly modified) from the example collection, originally taken from the previous work. A simple example containing a non-trivial lifetime error.

However, since there indeed is a lifetime error in this program, the compiler will report it and hence reject this program when one tries compiling it. The complete error message that is emitted by the compiler is given in Listing 3. It clearly states that the real issue is that the program does try to assign to `x` while it is still borrowed. The error message also points out that the assignment that is in question can be found on line 7. Furthermore, it states that the borrow of `x` does occur on line 3. However, this borrow will have to be used later on, especially later than line 7 (i.e. later than the assignment that is in question), to cause the actual error. The error message therefore states that the "borrow [is] later used" on line 8, by calling `take(w)`.

If one did not yet understand the entire error, this might be the point where one might ask oneself: "Is the compiler kidding me? Why `take(w)`?" More precisely, a programmer that is new to Rust will probably start wondering how this `w` is even related to the error, or more precisely to the borrow that was (apparently) created on line 3. There definitely is no `w` on line 3. The reader is requested to stop looking at the error message now. It does not provide an answer to this question(s). Hence, it lacks some information that is definitely needed for understanding the lifetime error in this code example.

Let us shortly go over the example and explain the cause for the error. On line 3, `x` is borrowed by `y`, due to the way the function `foo(p: T)` works. Then, on line 4, `y` is borrowed by `z`, since the function `bar(p: T)` works in the same way as `foo(p: T)`. Finaly, on line 5, the result of calling `foobar(&z)` is assigned to w, and since also `foobar(p: T)` works like

```
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

^^I^^Ierror: aborting due to previous error
```

Listing 3: Compiler error for Example 3. Additional warnings and references to the documentation are omitted. (This error message was created by the current stable Rust compiler version, that is 1.37.0, and the Rust edition was set to 2018, which is the default for this version.)

`foo(p: T)`, thereby `w` will borrow `z`. This chain can explain how that `w` is related to `x`. One could also state that, due to this lines, `w` "indirectly" borrows `x`. This suffices for explaining the cause of the lifetime error.

Conclusively, note that the compiler only reports the lines 3, 5 and 7 as part of its explanation for this error. However, in order to actually understand the error, one definitely also needs to consider lines 4 and 5 in addition to the lines that are mentioned by the compiler. Line 2 might also be relevant since it defines the initial variable. By this explanation we have already informally defined the information (as a set of line numbers) that is required in addition to the compiler error message for understanding the given error.

3.3 Long, obfuscated example

The previous example clearly demonstrates the insufficiency of the compiler error messages for understanding such lifetime errors. However, for an experienced programmer understanding this example will not be a complicated issue at all. Therefore, a more complex example will be presented now. In principle, this example is also based on the previous one. However, the interactions between the involved variables were diversified to reflect more possible reasons for the introduction of borrows, and quite a lot of extra code (that is not relevant for the actual error) was introduced. Therefore, the error is considered to be obfuscated, since we guess that also an experienced programmer will not find it on the first sight. The code for this example is given in Listing 4, and is also part of the example collection with the name `example3_long4.rs`.

```
1  fn main() {
2      let mut x = 4;
3      let y = &x;
4      let d = &x;
5      let y2 = move || {
6          println!("{}", y);
7      };
8      let y3 = y2;
9      let e = &d;
10     let mut g = 5;
11     let z = bar(&y3);
12     let f = &mut g;
13     let w = foobar(&z);
14     let mut a = 32;
15     let b = 42;
16
17     let s = &w;
18     let r = s;
19
20     x = 5;
21     *f = 42;
22     take(g);
23     take(w);
24 }
25
26 fn foo<T>(p: T) -> T { p}
27
28 fn bar<T>(p: T) ->T { p}
29
30 fn foobar<T>(p: T) ->T { p}
31
32 fn take<T>(p: T) { unimplemented!() }
```

Listing 4: Long, obfuscated example ('example3_long4.rs') from the example collection. An example containing a non-trivial lifetime error that was extended in such a way that the complete error is harder to spot.

The error message from the compiler (again using the stable Rust compiler version 1.37.0) is given in Listing 5. One can clearly see that it looks rather similar then the one for the simple example, and the reported error once again has the type E0506.[4] Also, when looking at the error one will wonder how that the lines 3 and 23 that are pointed out are actually related. So regarding the error the situation is similar. However, we will not go over this example in so much details for now, and hence not further explain the cause for the reported error. Instead, an explanation that is computed automatically by Rust Life will be presented as part of the evaluation in the section 5.2.

[4]This means that it indeed contains an error of the same type.

```
error[E0506]: cannot assign to `x` because it is borrowed
--> src/main.rs:20:5
   |
3  |     let y = &x;
   |             -- borrow of `x` occurs here
...
20 |     x = 5;
   |     ^^^^^ assignment to borrowed `x` occurs here
...
23 |     take(w);
   |          - borrow later used here

error: aborting due to previous error
```

Listing 5: Compiler error for the long, obfuscated example given in Listing 4. Additional warnings and references to the documentation are omitted.

3.4 Previous work

As mentioned before, this thesis is based on previous work by Dominik Dietler [2]. It already allowed acquiring information about a lifetime-related error. This includes all existing lifetimes in a method, all constraints in between of these lifetimes and partially 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 acceptable quality, but it was not clear if it was working fully flawless. Actually, some flaws were found (and fixed) as part of this thesis.

However, the existing code did pack all of this information 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 claim, we will shortly analyse the resulting graph for the simple example that was provided before as Listing 2. Note that its main methods only consists of 6 lines of source, and the existing work only considers the main method of this example. The graph that is emitted by the pre-existing tool for it is shown in Figure 3.1. 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 was not helpful for debugging the simple example.

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

A specific, relevant problem for this thesis is the actual visualization of 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 [10]. 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 [11].

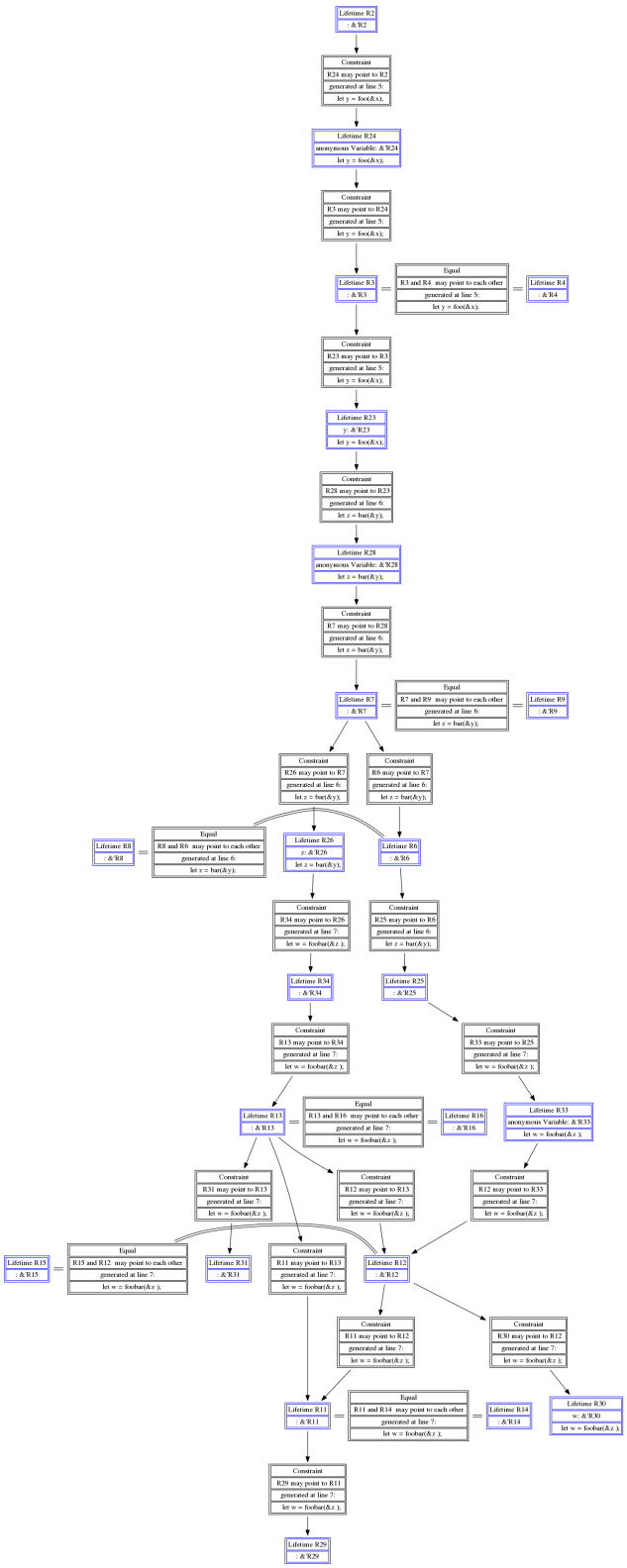

Figure 3.1: Resulting graph for the simple example, that was given in Listing 2.

Chapter 4

Explanations for Lifetime Errors

We'll propose an explanation of lifetime errors that solves some of the problems identified in the previous chapter. The final goal is to give a step-by-step guide as text that shall completely explain a lifetime error. This guide shall be created automatically by Rust Life, and should be easily accessible for a user through an IDE extension.

As a starting point Rust Life needs to get information about lifetimes (and especially lifetime errors) that exist in a given program. This includes the information about lifetimes and constraints that are given or directly derived from the source code, and also the information that is acquired by borrow-checking the program. Hence, we need both the inputs and the outputs for the borrow checker, or more precisely, of the naive implementation of Polonius. This information is already extracted from the Rust compiler by the previous work, by reading (textual) dumps from the compiler and by running the the naive Polonius implementation.

The previous work of Dominik Dietler aimed to compute the minimal set of Polonius input facts that is needed for explaining a lifetime error in a given example. The information it acquires thereby is then printed as a graph, yielding the graph that is given in Figure 3.1. This graph (or an internal representation of it) will be refereed to as "explain outlive graph" in the following sections. For getting this information, the previous work inverted the rules that are applied by the naive Polonius implementation (described in Matsakis's blog [3]). Then these inverted rules were applied to the output and input facts that Polonius produced, in order to get more information about the causes for a lifetime error that was found by Polonius before. In principle, the information about the found error that was returned by the borrow-checking by Poloinus before is used as a staring point for this process.

Even tough we decided to not keep using the inverted naive Polonius rules[1], we will still stick to operate on the input and output facts of Polonius. Especially, we will also use the error information that is given by Polonius as starting point. This directly implies that Rust Life will only be able to analyse and explain errors that are detected by the borrow checker. Therefore it cannot handle examples that contain any other error, and hence also not provide explanations for such examples. Note that also some errors that are (in some way) related to lifetimes (and related concepts) are not subject to be found by the borrow checker. An example are missing lifetime parameters. More information regarding the errors that are supported will be given in the evaluation chapter in section 5.3.

As mentioned before, the resulting output of the pre-existing code is not satisfying. In the next sections we describe how we solved various challenges that were needed in order to build the proposed explanation of lifetime errors. These challenges include finding a path in the explain outlives graph, simplifying this path and finding links from the graph to the relevant lines of source code.

4.1 A path trough a graph

The before-mentioned output is largely based on the so called outlives relation, that is an input fact for the naive Polonius borrow checker. Actually, it is a subset of this outlives relation, depicted as a graph. (Plus some extra information about relations to lines of source code and variables that will be explained largely in the next sections.) However, note that each constraint that is given in the outlives relation (basically each "edge" of the relation) is not depicted as a single (directed) edge in the printed graph, but instead is depicted as an extra node (a box labelled as "Constraint") giving extra information about it.

After studying these results for some examples, we noted that the lifetimes that seem to be relevant for explaining the lifetime error are aligned on a single path through this graph. This path is connecting a staring point and an ending points in the graph, i.e. a starting lifetime and an ending lifetime. In contrast, most other lifetimes that were displayed in the explain outlive graph were on some side branches that did not look like being of any relevance for understanding the lifetime error in the program. This observation also makes perfect sense when thinking about the information that is represented by the outlives relation, as well as the conditions that must be met in the case of a lifetime error existing in the program.

Given these observations, we decided to try finding this path through the explain outlive graph. As we have observed (for the examples we had) that

[1]The reasons that render using them unnecessary are given later in section 4.1.3.

there often is only one path throughout this graph, or any existing path seemed to contain all lifetimes that are of interest, we simply stuck to taking one path. Doing a search for a path in a graph, or more precisely in a relation between lifetimes, is not a hard task, and we decided to do a depth-first search. However, for actually searching a path one needs a staring and an ending node. (Or, in terms of the underlying relation, a staring and an ending lifetime.)

4.1.1 Where to start searching?

For getting a starting point we use the information that the naive Polonius borrow checker returns about an error it found. This information consists of a point and a list of loans.[2] Given this information, we started reasoning about the ways that these are related with the different facts that Polonius operates on. For this we extensively studied the rules that define the naive Polonius borrow checker. These rules were introduced in [3], described in the Datalog language [12]. These are essentially still used in the naive Polonius implementation. (Except some very small changes, and some recent additions in newer versions we do not yet use, since Rust Life sticks to a slightly older version of Polonius for now.) A write-up of all relevant rules that we created while studying them can be found in Appendix A.

It turned out that, by studying the output facts of Polonius (these are defined by the rules it applies), we can find a region that is essentially the one that causes the error. Basically, one could think that we do apply the rules `errors` and then `borrow_live_at` backwards, and then get a lifetime (region) and a loan (borrow) from the `requires` relation.

For now we are interested in the found lifetime, since it looks like the last one[3] in the explain outlive graph that we are interested in. So we wanted this lifetime to be the the last in the path that is computed. As we wanted to start the iteration (for searching) there, since it was the only lifetime we know at the beginning, we traverse the graph (or more precisely, the relation that defines it) backwards. By this we were able to start searching and constructing a path through this graph. However, we also need to know when the found path is complete. We need a way to determine that the found path does contain all needed information. Essentially, we need a termination criterion for our traversal.

4.1.2 When to stop searching?

It turned out that this was a rather hard problem, and we needed several trials to solve it. Finally, we came across the `borrow_region` input fact of

[2] Remember that a loan is now also called borrow.

[3] The last one that is encountered when traversing the graph along its directed edges.

Polonius. This provides a mapping from lifetimes (regions) to loans (borrows) and points. Recall that we also got a loan from analysing the output facts. As a side remark it can be noted that this loan actually is also always part of the list that is returned by the borrow-checking process as part of the error description. Rust Life does validate this as an integrity check. By searching the `borrow_region` relation for this loan, we get at least one lifetime. It turned out that this lifetime (Actually any of these, but in most cases there is only one.) is the first one in the explain outlive graph that is of interest. Hence, it is the one where the path we are searching for should start. Since we are searching backwards (i.e. from the end), the iteration will stop as soon as it hits any of these lifetimes. Hereby we got a termination criterion for our search. As soon as this criterion is met, Rust Life will stop searching the explain outlive graph and continue its operation using the path it found.

4.1.3 Is the "explain outlive graph" actually needed?

At this point, an implementation detail should be pointed out. The search implementation does actually not operate directly on the explain outlive graph, but instead it uses an internal representation of it. This is called the explain outlives relation, and as mentioned before (for the graph), it is a subset of the outlives relation, an input fact of Polonius. We therefore noted that the described search is actually walking a subset of a certain relation. Since the starting and ending points are given, and we are searching for one single path, we wondered if it would also work to do the search directly on the outlives relation.

If this would succeed, the computation of the explain outlives relation could be omitted. Recall that this relation is computed by the application of the inverted naive Polonuis rules. Note that this is a complicated process which is hard to validate. Its computation also takes some time.

Some simple evaluation has shown that it seems to work perfectly fine to use the outlives relation as input for the search of a path as it was described before. Therefore, we decided to use this instead, and thereby simplify the overall complexity of Rust Life. This process generates a relation that, when it is depicted as graph with the same means that were used before for depicting the explain outlive graph, seems to contain a lot of useful information for explaining the lifetime error in an input program. As an example, such a graph that depicts the result of this search when analysing the simple code example (given in Listing 2) is given in Figure 4.1.

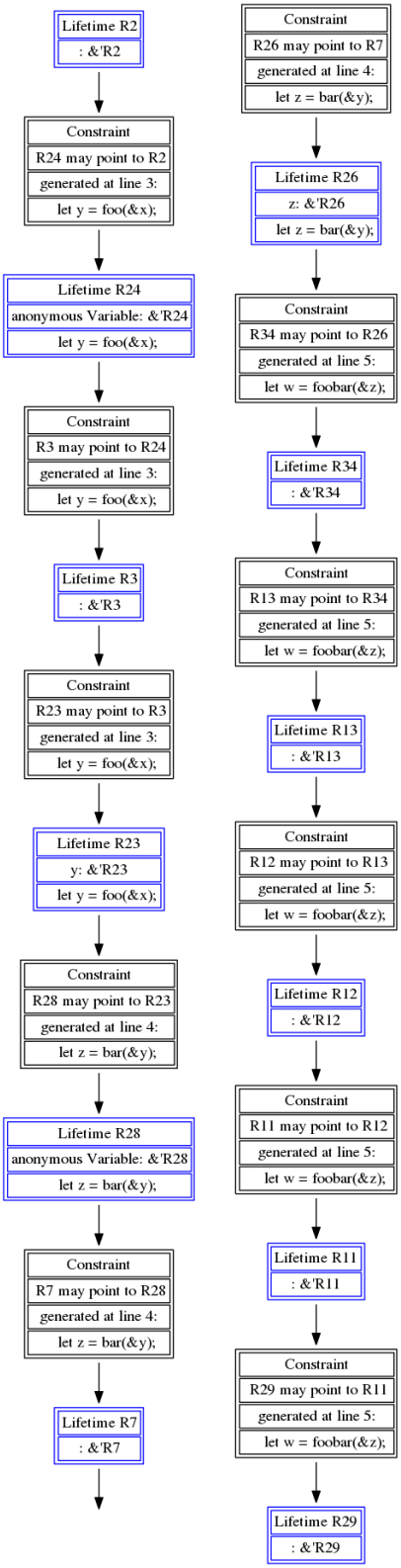

Figure 4.1: Result from searching the outlives relation for a path depicted as a graph for the simple example, that was given in Listing 2. The actual output figure was cut in the middle, and the two parts are shown side-by-side. One might note that this in fact is an extract from the graph given in Figure 3.1.

4.2 Linking lifetimes to source code lines

One can clearly see that the path, printed as a graph (given in Figure 4.1) looks less cluttered then the output we received from the previous tool, that was given in Figure 3.1. However, it still is rather large (especially long), and it also seems to miss some information. More precisely, the graph contains quite some lifetimes (blue nodes) that are not linked to a variable or a line of source code that this lifetime is related to. More precisely, for some lifetimes the graph states the (local) variable that this lifetime belongs to, as well as the line of source code that introduced this lifetime. However, for some other lifetimes this information is missing.

It might be the case that these lifetimes that were not linked to locals, and also not to lines of source code, simply represent lifetimes that are internals of the compiler. Hence, these are not related to any (local) variables, and it would make sense not to display them at all, since they would not be relevant for understanding the lifetime error. However, at this point we were not yet certain that this assumption about these lifetimes actually holds. Therefore, we first wanted to check (and try to improve) the linking of these lifetimes to variables and source lines.

4.2.1 Linking to locals by using MIR

In order to get a linking from lifetimes to local variables (locals) the previous code does parse a dump of the MIR (a compiler-internal program representation) that was emitted by the Rust compiler. (The compiler instance that runs as part of Rust Life is instructed to create this dump.) Unfortunately it turned out that the parsing was not working in a completely flawless manner, and therefore it missed some linkings. In addition there also was a flaw in the internal representation of the acquired information, that also could have lead to a loss of some linkings.

Since parsing a dump of the MIR is rather cumbersome, error-prone and generally not portable across compiler versions we were trying to get rid of it. As alternative we tried to get a copy of the internal MIR structure of the compiler instance that is used by Rust Life. At the point where the actual analysis is invoked (technically, as a callback from the compiler) three different versions of MIR (At different compiler-internal stages) are accessible by using the compiler-internal API. Unfortunately, after some trying it turned out that all of them represented a state that was too late (in the compiling process) for our purposes. None of these versions preserved the lifetimes together with their identifiers that were used at the stage of borrow-checking the program. But the only way we can identify the lifetimes in the facts of the borrow checker is by using these identifiers. Therefore, the lifetimes that define the path we found are also specified by these identifiers. So all MIR

versions that were accessible by using the compiler's internal API turned out to be unusable for Rust Life.

Since we did not want to apply any changes to the actual compiler (this had exceeded the scope of this thesis), we decided to stick to parsing a dump of the MIR that was created at a stage when the needed information was still around. We re-implemented the parsing of the lines that really seem to be of interest for Rust Life in a simpler way and adapted the internal representation to meet all our needs. By using this information, we were able to get information about locals for some more lifetimes. And from the information about the locals it is also possible to get the information about a related line of source code. This information is then included in the graph representation that is emitted.

4.2.2 Using point information from "borrow_region"

After applying this improvement it still seamed that some information is missing. More precisely, for some examples the first node in the found path was not connected to a local, which we considered as a potential issue that should be resolved. In the previous section the relation `borrow_region`, an input fact of Polonius, was used to link loans to lifetimes. However, remember that this relation also contains information about (program) points. Therefore it can also be used to map from some lifetimes (the ones it contains an entry for) to points.

Additionally, the previous work contained some logic that allows to map from a program point to a line of source code in the input program. This logic is used for getting the line of source code that is included in the graph as extra information for constraints. Therefore, the point that is given for each edge in the outlives relation is used.

By applying this mapping logic to the point that is given for some lifetimes by `borrow_region` it is possible to link these lifetimes to a line of source code that seems to be sensible. More precisely, this method allows to get a helpful, sensible line of code to be displayed next to the first lifetime (node) in the path for a notable number of examples. Therefore, this information is now also included when printing a path as graph. More precisely, there might be multiple lines that would be included, since `borrow_region` could map one lifetime to multiple points. However, we never found an example were this case actually occurs.

For completeness, we want to point out that we also tried to apply this mapping by using the `requires` output fact of Polonius instead of `borrow_region`, since it also gives a relation form lifetimes to loans and points. However, by this change we roughly got a list with all lines that are spanned by a given lifetime. This also seems sensible when reasoning about the meaning of the

`requires` relation. However, since this information seemed not to be really helpful (and even rather superfluous or confusing) when it was included into the graph, we dropped this approach and kept using the information from `borrow_region`.

An output graph that shows the path with this extra information for the simple example (given in Listing 2) is given in Figure 4.2. While implementing these changes, also some internals of Rust Life were changed and refactored. (This was also needed as preparation for further steps and improvements.) However, these technical changes only cause very small differences in the graphical representation. Actually, the only changes are that all printed lines of source code are now prefixed with their corresponding line number and some small improvement in the graphical alignment of their text. All other differences between Figure 4.2 and the previously given Figure 4.1 are due to the improvements that were described in this section. (Particularly well notable at the first and at the last node.)

4.3 Optimize the path aka get rid of the compiler's internals

Despite the optimizations of the linkage from lifetimes to the corresponding locals and source lines, Figure 4.2 still contains quite some lifetimes that are not accompanied with information about a (local) variable or a line of source code. However, for quite some examples the graph contains a node (lifetime) for all lines that seem to be relevant for understanding the lifetime error. Therefore, all of these nodes that are not linked to a (local) variable are considered to be compiler internals and therefore are not relevant for explaining the lifetime error. Hence, these should not be shown as part of the final output.

Furthermore, there are also some lifetimes that were linked to a so called "anonymous variable". These are variables that are not created explicitly in the program, but are introduced internally by the compiler. Therefore they also do not have a name. Since the nodes that are not linked to such anonymous variables seem to provide all needed information (i.e. there seems to be one lifetime for each needed line), these lifetimes linked to anonymous variables should also not be included in the resulting output.

Therefore an extra step of improvement was implemented. This step will walk the graph (technically given as a relation between lifetimes) and remove all nodes (lifetimes) that are either not liked to a (local) variable or are linked to a (local) variable that has no name. For doing so, it will use the information that was acquired for enriching the graph by the means that are described in the previous section. However, this step will always preserve the first and the last node (lifetime) of the path. This is due to the fact

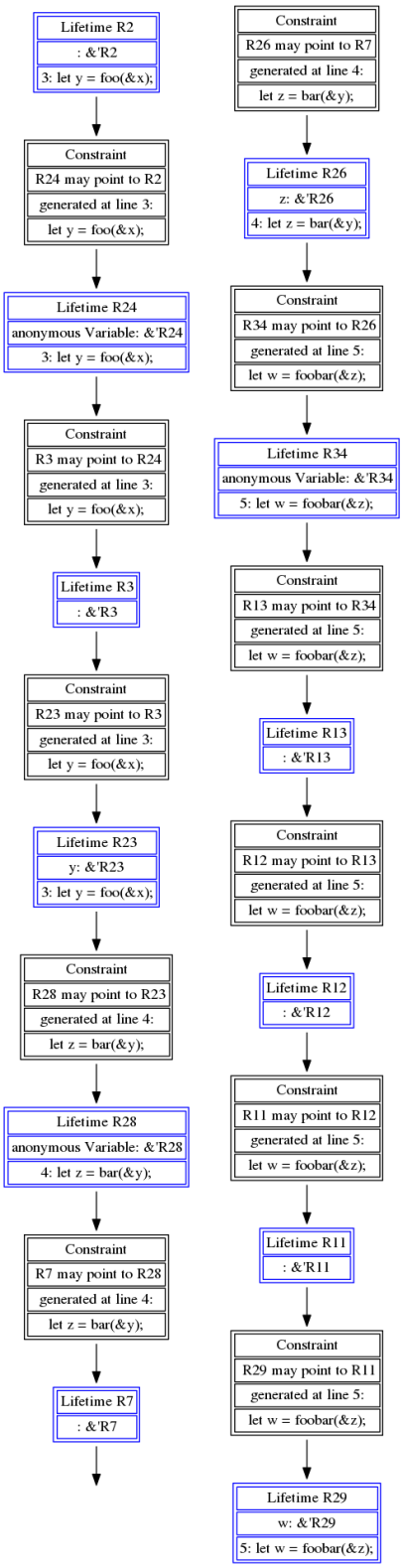

Figure 4.2: Once again, the path for the simple example (given in Listing 2) depicted as graph, now with the extra information that is acquired by the improved linking of lifetimes to source code lines. Also, due to some internal changes in Rust Life, all reproduces lines of source code are now prefixed with their respective line number. The actual output figure was cut in the middle, and the two parts are shown side-by-side.

that these nodes seem to be always necessary for understanding the lifetime error, but in some cases they cannot be linked to a local variable. Still, especially the first node can often be linked to a line of source code by using the `borrow_region` relation and the mapping from program points to source lines. So there is some usefull information that can be displayed as part of this node.

The final result for the simple example, after applying these optimizations, is given in Figure 4.3. This might be a good point to stop reading and look at the output graph. Then, put it next to the source code of the simple example. (that is given in Listing 2.) By looking at both of them, applying the given linking from the graph to the source code and then following the given path it should become quite clear why the compiler is stating that `take(w)` is a use of a borrow of `x`. (Rember the error message given in Listing 3.) So one should note that this graph is giving useful help for understanding a lifetime error in a given Rust program.

That said, now we reached a point where Rust Life creates a usable visualization of a lifetime error in Rust. It should also be noted that it seems to contain all information that is needed for understanding the error, when the compiler error message is given as context. We therefore decided to use this graph as the final output of Rust Life, at least for the scope of this thesis. However, running Rust Life from the command line for each program, and then looking for the resulting graph in a certain directory where one finds a simple dot file (that requires extra software for viewing) is not that comfortable.

4.4 Making it accessible for users: an IDE extension

In order to offer an easy way to use Rust Life, we proposed the creation of an IDE extension that will run it on Rust programs and display the resulting visualization next to the editable sources. We decided to realize this as an extension for Visual Studio Code. (This allowed us to reuse code of existing extensions with a similar workflow.) It is called "Rust Life Assistant".[4]

The extension is rather simple and implementing it did not include solving any significant new problems. What it does is first running Rust Life on the file that is opened in the active editor window.[5] Rust Life was extended to emit its error explanation information as JSON dump into a fixed file.[6]

[4] At the time of writing it is not available on the Visual Studio Code Extension Marketplace.

[5] It will abort if this is not a Rust source file.

[6] Form an implementation view, this is done by serializing an internal structure that contains all needed information using the "serde" library. This is very simple and ensures that the created JSON has a consistent structure.

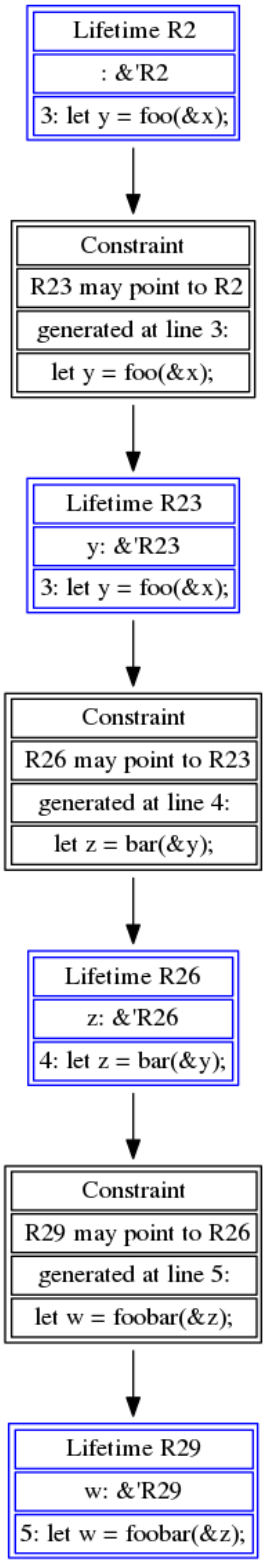

Figure 4.3: The final resulting graph (showing a path) for the simple example. Looking at this, the source code of the simple example (see Listing 2) and the Rust compiler error message (given in Listing 3) should be sufficient for explaining the lifetime error in this example.

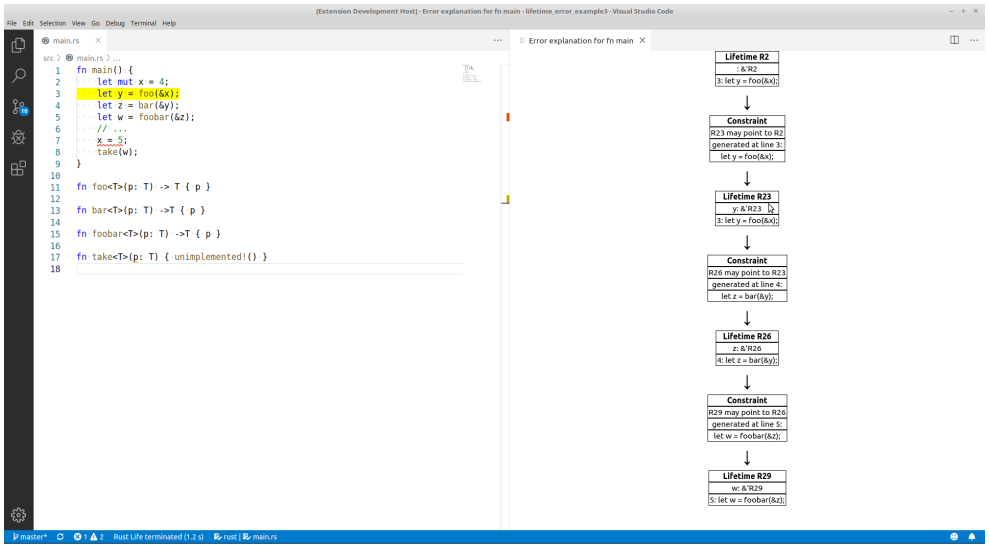

Figure 4.4: Rust Life Assistant showing the graph and a highlighted line in the source code. The highlighting was triggered by a click at the position of the included mouse pointer. The editor is showing the code of the simple example, and hence the same graph as given in Figure 4.3 is displayed. We used a light colour theme and the highest possible zoom level for creating the screenshot.

Once Rust Life is done, the extension will read in the emitted JSON from the file.[7] Finally, this information is used to display the path that shall explain the lifetime error as a graph to the user. So Rust Life Assistant will create a graph as HTML (that is then shown in a WebView) from the information it gets from Rust Life. This means that it does not use the dot file that is created by Rust Life.

Since all lifetimes (nodes) and also all constraints (also displayed as nodes in the graph) are linked to a certain line of source code, we also wanted to make this linkage visible in the IDE extension. Therefore we made the nodes of the graph click-able. As soon as a user clicks on one of the nodes of the graph, the line of source that corresponds to it will be highlighted in the editor that shows the input source code. This allows the user to easily find the line that a certain lifetime (or a certain constraint) is coming from. A screenshot of the IDE extension can be found in Figure 4.4. It also shows the highlighting of a line, triggered by a mouse click, at the position where the pointer is displayed.

At the time of writing we have a working prototype of the extension. Its "README.md" is given in Appendix B.

[7] Since the extension is written in TypeScript, that is a superset of JavaScript, doing so will create an object that can be used as such.

4.5 An alternative to the graph-based visualization: simple text

The graph that is displayed is certainly helpful. However, understanding it and extracting the needed information from it might take some time for a programmer. Therefore, we will present another approach that we tried out as an extension in this section. This is explaining the error with text, which was also implemented as part of the Rust Life Assistant extension. Hence, the representation is created by the extension, based on the information that it gets from Rust Life as JSON.

As a starting point, the (original) error message from the Rust compiler is displayed. This is sensible since the information we present is actually an addition to it. In a next section, there is a simple ordered list, featuring some points for explaining the lifetime error.

The first point is always giving the information from the first node in the graph, stating that this variable is borrowing something that we call "the initial variable". This means the variable that is initially causing the lifetime error, e.g. for the simple example this would be `x` that is used while being borrowed. This point also contains the line that is part of the first node.

Then, Rust Life Assistant iterates over the edges of the graph. These are represented by the black nodes in the graph printed by Rust Life. For each edge, a new point is added. It first states that the variable of the end node (of the edge) borrows the variable of the start node (of the edge). As cause, the line that was reported as causing the handled edge is reported. Unfortunately, this leads to edges that stated that a variable is borrowing itself. Since this is redundant, useless information, Rust Life Assistant will not include such lines. Hence, they are omitted from the list.

Finally, the last point will state that the variable from the last node in the path is later used. This should also be the variable that will be mentioned on the last line that is reported as being part of the error cause by the compiler. Therefore, the title printed before the ordered list also states that the following will be an explanation for why that the variable that belongs to the last node is borrowing the initial variable.

In the list, all variable names and all included lines of source code are clickable. Therefore their text is blue. By clicking a variable the user can request highlighting of the line of source code that this variable belong to. Or, if a source line is clicked, its equivalent in the text editor is highlighted.

Also this operation mode is a working prototype. A screenshot showing this operation, also including the highlighting of lines of source code by a click by the visible courser, is given in Figure 4.5. Listing 6 shows the text of the list (including the title) that is emitted when analysing the simple example.

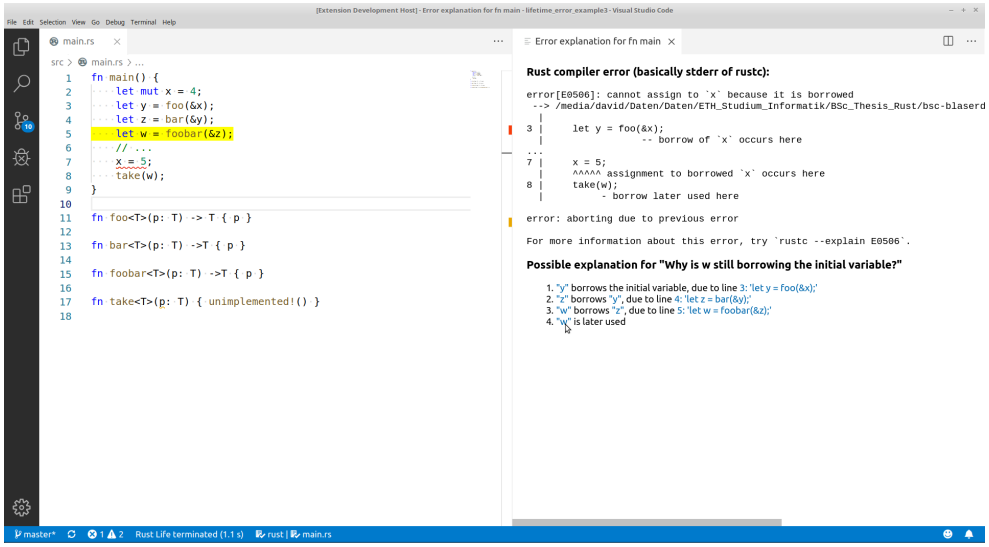

Figure 4.5: Rust Life Assistant showing the text based error explanation and a highlighted line in the source code. The highlighting was triggered by a click at the position of the included courser. The editor is once again showing the code of the simple example. We used a light colour theme and the highest possible zoom level for creating the screenshot.

```
Possible explanation for "Why is w still borrowing the initial variable?"
1. "y" borrows the initial variable, due to line 3: 'let y = foo(&x);'
2. "z" borrows "y", due to line 4: 'let z = bar(&y);'
3. "w" borrows "z", due to line 5: 'let w = foobar(&z);'
4. "w" is later used
```

Listing 6: The textual error representation that is emitted by Rust Life Assistant for the simple example. (Code given in Listing 2) Not including the compiler error message. The complete output of the extension is visible in Figure 4.5.

Chapter 5

Evaluation

In this section we will evaluate the prototype that was introduced in the previous chapter. This evaluational serves to asses the tool and especially its output for a variety of examples. This means that we want to check which examples are supported well by the tool This will show for which errors it can provide helpful explanations. Therefore, an evaluation using Rust Life Assistant[1] was done by testing it for a set of examples that we collected before.

5.1 A set of examples

In order to test the prototype we need examples to run Rust Life (Assistant) on. This means that we need Rust Programs that contain a lifetime error. More precisely, it must be a lifetime error that is found by the borrow checker, since Rust Life only handles these errors. In the rest of this section we will describe the sources from which we obtained the examples, some of which were also used during the development of the tool.

As a staring point we had nine examples from the previous work, but unfortunately some of these do not actually fall into the scope of Rust Life. These examples are called "example{x}.rs", where x is the number of the example. Furthermore, we got examplse by some other ways.

The Rust Compiler Error Index [13] It provides at least one example for each error code that is emitted by the compiler. Hence, this index provides code samples that contain errors. It apparently contains the same texts that

[1]Since Rust Life Assistant directly uses Rust Life, using it for testing will also sufficiently test Rust Life. The only part that is not tested by this is the printing of the graph as dot file by Rust Life, which is acceptable. (Especially since this part is very similar to the creation of the graph-based visualization by the IDE extension.)

are also printed when the `--explain` option of `rustc` is used.[2] These examples work well to determin if Rust Life does operate correctly when faced with a certain type of error. Thereby, a type of error means an error that causes a certain error code. By this one can mostly check that Rust Life will not crash for such an error and not provide any unexpected output.[3] However, since these example form the error index are inteded to explain these errors to (novice) programmers, all lifetime errors that these examples contain are rather trivial, and the Rust compiler error message sufices for understanding them. Hence, the assistance of Rust Life would not be needed for understanding these examples. The files with the snipets that we copied from the error index are called "EI_{`error_number`}_ex{`num`}.rs", where `error_number` is the number of the error (including the starting letter 'E'), and `num` is the number of the example.[4]

Writing examples manually Another way that we used to obtain samples for testing was constructing them manually. Some of the examples where essentially created from scratch, mostly with the goal of including a certain language construct for testing, e.g. loops or conditionals. These examples are stored in files that have names starting with "example", followed by an underscore and then giving a keyword for the construct that they feature. In the end their names provide some kind of (non-uniform) numbering for different examples. These with similar numbers are often rather similar.

Another approach we applied is extending existing examples. Thereby we either added more levels of indirection in the erroneous chain of borrows. This causes the errors to get more complicated and therefore Rust Life can shine when giving an explanation while the compiler error message is no longer sufficient for understanding the error. Another change that we applied more rarely was to add more lines of source code that were not related to the error. This will make finding the ones that are relevant harder, and thereby allows for showcasing the usefulness of Rust Life. Especially if the non-related lines do also touch variables that are relevant for the error in ways that do not influence the actual lifetime error.[5]

[2]Unfortunately, the index is not always complete and up to date. For some error codes, no information is avilable.

[3]Note here that the error code is not the only way in which examples can differ, there are more options. Hence, it can happen that Rust Life succeeds for one example with a certain error, while it fails for another example that contains an error with the same error code. So covering all error codes does not provide a full coverage over all relevant aspects of examples.

[4]Starting from 0, in most cases the one with number 1 is a slightly modified version that was created to cope with a former limitation of Rust Life that was resolved later. These examples are marked accordingly with an asterix in the Table 5.1

[5]Unfortunately, such addition sometimes also lead to the disclosure of some issues of Rust Life. More precisely, such changes caused that Rust Life will include some unrelated, and hence superfluous information into its explanation.

The extended examples from the error index are marked by the suffix "_ext" and a number as addition to their name.[6] When examples from the previous work were extended, some text (or at least a number), preceded with an underscore, was added to their name. (This was done for examples 3 and 9.) Last but not least, a special example is "find_path_in_graph_ex0.rs". It was created for testing the process of finding a path through the outlives graph (or relation).

One other example This example is called "stackoverflow1.rs" and was obtained from a question on Stackoverflow that caused a lifetime error that seems to be hard to understand. (Unfortunately, this example also seems to confuse Rust Life.) Last but not least there is the hand-crafted "no_error_example1.rs". As the name states, this example does not contain an error, and it therefore is an expected behaviour that Rust Life will not create any error-explanation-output for it. Hence, this example does not fall into the scope of Rust Life, but it shall be noted that Rust Life does not crash when it is run on this example.

5.2 Rust Life Assistant in action

The evaluation was done by running Rust Life Assistant on each example, both generating the graph-based visualization and the textual error explanation. This process will be explained step by step in the next paragraphs by applying it to the long, complex example that is given in Listing 4.

The very first step is of course opening the file in Visual Studio code.[7] Then, we executed the Visual Studio Code command `Rust Life: Visualize as Graph` to create the graph based visualization. As reference, the graph output for the long, complex example is given in Figure 5.1.[8] Now we looked at the nodes and then at the lines that they were mentioning. By doing so for the graph given in Figure 5.1 and the code in Listing 4 one will note that this will be helpfull to understand the compiler error message that was given in Listing 5. This process is quite simple when it is done by using the IDE extension, since lines can be highlighted by clicking on graph nodes.

[6] A special case is the example file that was named "EI_E0713_ex0_refmt.rs", it was reformatted to make its code more readable. By doing so the compiler error message got more readable, and the output of Rust Life (Assistant) also was slightly improved. This is probably mostly due to the fact that both of these reason about complete lines of source code in their explanations.

[7] More precisely, in an instance of Visual Studio Code that runs the Rust Life Assistant extension. For simplicity we simply used the "Extension Development Host" instance that was running the extension (in debug mode).

[8] The Figure actually shows the graph that is emmited by Rust Life as dot file, and not a screenshot of Rust Life Assistant, since this graph suits better for being shown as part of a report.

Since the information that the graph contains for this example is sufficient to complete the error mesage and no superfluos information is included, the graph-based representation passed the test for this examples.

The next step consists of testing the textual representation. This can be run on the example opened in Visual Studio Code by running the command `Rust Life: Explain with Text`. The result of doing so for the long, complex example is given in Figure 5.2. We then read the explanation point by point, check the mentined lies of source code and thereby can see that all information that is needed for understanding the error message is contained. Since it also does not contain any superfluous lines, the textual error explanation also passed the test for this example.

This process was also applied for all other examples that we collected. The next chapter presents the results.

5.3 Results

The results of the evaluation for each example are given in the Table 5.1 The first column gives our judgement for the error message that is given by the Rust compiler.[9] If the error message is fine (indicated by ✓) it is self-contained and sufficient for explaining the error in this example. In this case Rust Life is not really needed for this example, as the lifetime error that it contains is rather trivial. For this examples, the output of Rust Life (Assistant) will usually be small, and not provide much extra information.[10] This is fine, since there is not need for extra information. The next column states our assessment of the graph output of Rust Life Assistant, and the last column gives the evaluation for the text-based explanation.

In the table the following symbols are used: A checkmark (✓) indicates that the output is fine for this example and the test passed. In contrast, a cross (✗) indicates a failure, e.g. Rust Life (Assistant) does not create any output or the resulting output is insufficient. If these signs are surrounded by parentheses, this indicates an addition of "partially" (I.e. either "partially passed" or "partially failed".) We use "partially passed" when clearly more then half of the output is correct, but it still contains issues. In contrast, "partially failed" indicates that the result contains small bits of good information, but it is seriously flawed. The letter 'S' indicates that Rust Life did include superfluous information (nodes in the graph, points in the textual explanation, both usually related to superfluous lines of source code). This is not a complete failure, but since it can confuse users it should also not

[9] Using the Rust compiler from the toolchain "nightly-2019-05-21". This is the version that is also used by Rust Life.

[10] Usually, the graphs consist of only two lifetimes, and the textual representation includes two or three points.

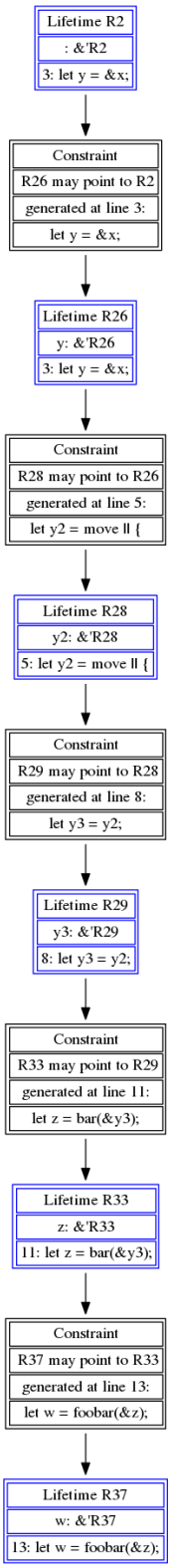

Figure 5.1: Resulting graph (output of Rust Life) for the long, complex example, that was given in Listing 4.

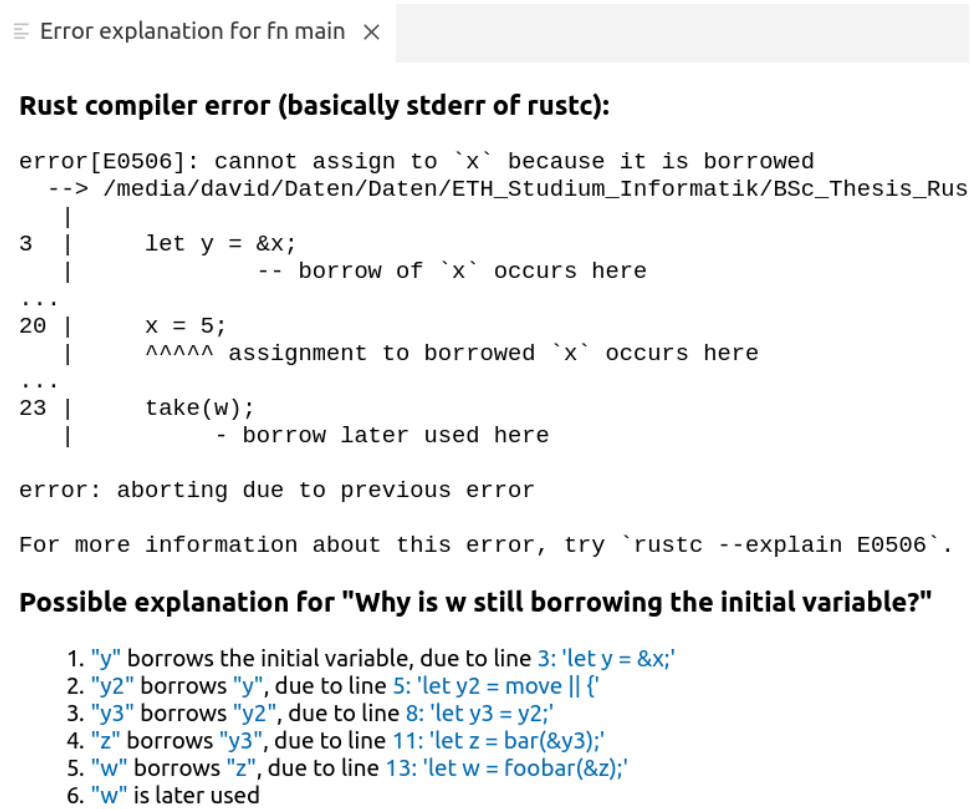

Figure 5.2: Resulting textual explanation (output of Rust Life Assistant) for the long, complex example, that was given in Listing 4. Note that this screenshot only shows the relevant part of the right panel of Visual Studio Code to increase the legibility.

be considered as a complete pass. Finally, the $\sim$ symbol indicates that the example is not in the scope of Rust Life and hence it cannot be used for testing. (In most cases the error that is contained in such an example is not supported. Probably these errors are not subject to be found by the borrow checker.)

The asterisk for "example3_long3.rs" indicate a very special case that will be discussed in the next section. Some results were marked with the superscript 'V'. This indicates that in the output for this example some variables (in the textual representation, equivalent to lifetime nodes in the graph) were linked to the source code lines where they were defined, and not to the one that the relevant assignment to them occurs on. This does happens in most cases when variables are defined on a different lines then the relevant assignment is happening on. However, this is not considered as being an issue. Finally, an asterisk after the name of a file from the error index indicates that this example is essentially the same then the one in the previous row. Its code was only altered in insignificant ways to allow using this example

while Rust Life still had some limitations that were resolved by now.

5.4 Explanation and assessment of the results

Rust Life (Assistant) works fine for most examples from the error index, and also for the ones that were extended. The biggest exception is "EI_E0713_ex0.rs", the explanations that are provided for it are quite messy. (However, the compiler error message for this example is also not that well-readable.) Therefore, the code of the example was formatted in a way that should make it more readable, yielding "EI_E0713_ex0_refmt.rs". For this example we do not only get a clearer compiler error message, but also the output of Rust Life gets more readable. Still, there remains a node (in the graph) that is not liked to a source line, and therefore the textual representation refers to a variable that has no name. So Rust Life seems to have problems dealing with this example. There also is a limitation in the textual explanation for "EI_E0716_ex1.rs". A trial to parse a line of source code for getting a variable name that went wrong leads to the usage of a function name as a function name in the explanation. This is (of course) wrong, but is probably not specific to an error with this error code. Instead, this uncovers a general flaw in the parsing of source lines to get variable names.

A significant issue is discovered by the examples with names starting with "example_loop_2", that feature some examples with a loop over a very simple linked list. Rust Life does not produce any output at all for this examples, apparently it is not able to find the error in these examples. Unfortunately, we cannot explain this behaviour. There are other examples that contain an error with the same error code that are handled just fine by Rust Life. This issue is especially annoying since extra help (in addition to the compiler error message) would be necessary for understanding the issue in one of these examples. Further work would probably need to study these cases in much detail, since these examples could direct to an issue that is rather deep in the system.

For "example3_long3.rs", as well as "example3_long3_err0.rs" and "example3_long3_err1.rs" the graph does contain extra lifetime nodes (and hence, extra lines) that are not really needed for understanding the error in the examples. These lifetimes (and lines) are not completely unrelated to the error, since they do indeed touch variables that are involved in the cause for the lifetime error. Still, they should probably not be included in the output. For the last two examples these superfluous lifetimes then lead to extra points in the textual representation. Due to the way that these points are constructed from the graph data they contain statements that are not really sensible, and definitely should not be included. Note that no such extra lines are included in the textual explanation for "example3_long3.rs", but this only happens by

Name of example:	Compiler:	Graph:	Text-based:
EI_E0499_ex0.rs	✓	✓	✓
EI_E0499_ex0_ext0-0.rs	✗	✓	✓
EI_E0499_ex0_ext0-1.rs	✗	✓	✓
EI_E0499_ex0_ext1-1.rs	✗	✓	✓
EI_E0499_ex0_ext1-2.rs	✗	✓	✓
EI_E0500_ex0.rs	✓	✓	✓
EI_E0500_ex1.rs*	✓	✓	✓
EI_E0500_ex1_ext0.rs	✗	✓	✓
EI_E0500_ex1_ext1.rs	✓	✓	✓
EI_E0501_ex0.rs	✓	✓	✓
EI_E0501_ex1.rs*	✓	✓	✓
EI_E0501_ex1_ext0.rs	✗	✓	✓
EI_E0502_ex0.rs	✓	✓	✓
EI_E0502_ex1.rs*	✓	✓	✓
EI_E0502_ex1_ext0.rs	✗	✓	✓
EI_E0502_ex1_ext1.rs	✗	✓	✓
EI_E0503_ex0.rs	✓	✓	✓
EI_E0504_ex0.rs	✓	✓	✓
EI_E0505_ex0.rs	✓	✓	✓
EI_E0506_ex0.rs	✓	✓	✓
EI_E0597_ex0.rs	✓	✓	✓
EI_E0597_ex0_ext0.rs	✗	✓	✓
EI_E0713_ex0.rs	(✓)	(✗)	(✗)
EI_E0713_ex0_refmt.rs	✓	(✓)	(✓)
EI_E0716_ex0.rs	✓	✓	✓
EI_E0716_ex1.rs	✓	✓	(✗)
example_clousures1-1.rs	✓	✓	✓
example_clousures1-2.rs	✗	✓	✓
example_clousures1-3.rs	✗	✓	✓
example_ifelse1.rs	✗	$✓^V$	$✓^V$
example_ifelse2.rs	✗	$✓^V$	$✓^V$
example_loop_1.rs	✗	✓	✓
example_loop_2-1.rs	✓	✗	✗
example_loop_2-2.rs	(✓)	✗	✗
example_loop_2-3.rs	✓	✗	✗
example_loop_3.rs	✓	∼	∼
example1.rs	✗	$✓^V$	$✓^V$
example2.rs	✓	✓	✓
example3.rs	✗	✓	✓
example3_long.rs	✗	✓	✓*
example3_long2.rs	✗	✓	✓
example3_long3.rs	✗	S	✓
example3_long3_err0.rs	✗	S	S
example3_long3_err1.rs	✗	S	S
example3_long4.rs	✗	✓	✓
example4.rs	✓	✓	✓
example5.rs	✓	✓	✓
example6.rs	✓	✓	✓
example7.rs	✗	∼	∼
example8.rs	✓	∼	∼
example9.rs	∼	∼	∼
example9_2.rs	∼	∼	∼
example9_3.rs	✓	✓	✓
find_path_in_graph_ex0.rs	✗	✓	✓
no_error_example1.rs	∼	∼	∼
stackoverflow1.rs	(✓)	(✓)	(✗)

Table 5.1: Results of the evaluation for each example. The meanings of the used symbols is explained in Section 5.3.

coincidence, due to a implementation detail. Therefore, this result is marked with an asterisk, since it is just a lucky coincidence that this example passes the test. We speculate that this issue is caused due to the way that the search for a path through the `outlives` relation is conducted. More precisely, we assume that it is caused by just taking the first path that is found. This could probably also explain the reason for not having this issue with other examples that are extramly similar, like "example3_long2.rs" or "example3_long4.rs". (The last-named only differs from "example3_long3.rs" in one signle line.) We therefore propose to try fixing this issue by alwyas searching for the shortest path that can be found. An alternative option might be to search for circles in the path by matching on the related variables and then remove any such circles. However, this are just some first ideas that were not yet tested and would neede further investigation, especially scinethe they could eventually also cause new problems.

The examples 7 and 8 from the previous work seem to not contain an error that can be handle and analysed by Rust Life. In contrast, "example9.rs" contains two errors. Rust Life does not support handling examples with multiple errors. (It might work for some such examples, but this was not tested, and therefore we will not use this example for the evaluation.) This also applies to "example9_2.rs". However, in "example9_3.rs" we succeeded in altering the original "example9.rs" in a way that only one error remained in it that was handled by Rust Life successfully.

The example from the Stackoverflow post is rather confusing. The graph representation seems partially sensible, whereas the textual representation for it looks quite messy. We will not go into more details about this example. Investigating it would certainly take a lot of time.

Conclusively, one can note that Rust Life and the Rust Life Assistant work fine for a good portion of these examples. Despite Rust Life (Assistant) working fine for most examples, the evaluation also uncovered some limitation.

5.4.1 Known limitations

The following limitations and issue can be classified and listed:

- Rust Life fails to operate on some examples that contain loops. (We cannot determine if this is due to an implementation flaw or due to a limitation of our approach.)
- For some examples, Rust Life reports superfluous information. This seems to happen for long, complex example. Especially for examples that contain variables that are involved in the lifetime error, but there are also operations on these variables that do not affect the lifetime error. Some speculative explanations for this issue were given before.

- The parsing of lines of source code for getting names of (local) variables is not working completely flawless, in some cases some part of the source code that is not a variable name is used as such. It is not clear if it would be possible to always get the name of the desired variable by parsing a single line of code. Since this approach is also more of a "workaround" it might also be worth trying to get rid of it and replace it by a solution that is more stable.
- In some cases Rust Life fails to find a corresponding (local) variable and also a corresponding line of source code for a lifetime. This is sometimes visible at the last node of the path, since it will never be removed from the path while optimizing it.
- If Rust Life fails, Rust Life Assistant might display old results from a previous execution. This is a simple implementation insufficiency that could be resolved by a simple clean-up step.[11]
- Rust Life (Assistant) can only operate on single files with Rust code. Especially it cannot deal with complete crates that are compiled by cargo. (Fixing this is probably not hard, but would include some work.)

Additionally, as pointed out multiple times before, Rust Life can only handle errors that are subject to be found by the Polonius borrow checker. We are not certain if there is a fixed set of error codes that belong to errors that will always be found by Polonius. It is certain that Rust Life can handle some (probably even most) examples that contain an error that leads to an error code that Table 5.1 list a specific example for. However, there are also some other error codes that clearly affect lifetimes that are not handled correctly by Rust Life.[12] We therefore consider this to not be in the scope of Rust Life. Still, not being able to handle this could be a limitation. Unfortunately, we also cannot state for certain that these errors are never subject to be caught by Polonius, since we cannot exclude that Rust Life fails to handle these due to an implementation flaw. E.g. there might be an issue in parsing the dump that gives the Polonius input facts that will be processed by Rust Life.

5.5 Self-assessment of the implementation and code quality

5.5.1 Rust Life

Rust Life is a working prototype tool, with the limitations identified in Section 5.4.1. This does also apply to the code quality, it is probably best de-

[11] The main reason for not yet implementing this is to allow for simple debugging of the interfacing between Rust Life and the IDE extension.

[12] At least for the examples with such error code that we tried.

scribed as "research code". We also indicated before that we cannot really asses the coverage of our testing by examples.[13]

However, we tried to structure the code into methods that do have a reasonable complexity, and we also wrote extensive documentation for most of these methods. We therefore believe that the code is (mostly) understandable and can also be modified or extended within a reasonable amount of time. One of the main issue is that there are quite some leftovers from different old ideas that we implemented, which are no longer needed.[14] This also causes Rust Life to produce several different outputs that represent the state at different stages of optimizations that are now implemented.[15] Also, compiling Rust Life will cause the compiler to emit some warnings. These mostly point out unused variables and methods that could either be removed or their names could be prefixed with an underscore. Applying these rather simple changes it would probably be possible to significantly increase the code quality of Rust Life and make it more readable.

Finally, it should also be pointed out that Rust Life depends on a slightly old nightly build of the Rust compiler toolchain. More precisely, this is the version that is identified by `nightly-2019-05-21`. Using a nightly toolchain is not evitable since Rust Life needs to access the internal API of the compiler. However, it would be necessary to update it to a never version sometimes to ensure that Rust Life can also handle new features that are added to Rust.[16]

5.5.2 Rust Life Assistant

Rust Life Assistant is a prototype IDE extension. Even tough some approaches that are used in its code are definitely not good style, we still ended up trying to give at least some sort of structure to the code.[17] Since the code is still rather small and we also added some documentation we are confident that it is still readable and understandable.

To use the extension in production, we suggest to apply the following changes: One should probably restructure some of the code and some parts should possibly even be completely rewritten. A good example is probably one

[13] We do not have any automated testing framework and also did not come up with unit tests, since the units of Rust Life are rather complex and have rather complicated (external) dependencies.

[14] At least, when one is only interested in the output of Rust Life.

[15] These extra outputs were helpful for debugging and testing at some stage and also for writing this report. It would be rather easy to remove the outputs. Then one could also remove the logic that is no longer used by following warnings of the compiler that point out unused variables and methods. Doing so is probably easy.

[16] The process of updating can sometimes be cumbersome and time-consuming, since the internal APIs of the compiler are unstable and might change. In the worst chase, such changes could even cause breakage in Rust Life.

[17] This was quasi inevitable when we were implementing the text-based error explanation.

function that creates an entire HTML document that will be shown in a WebView by concatenating strings. It even includes an in-line script into the HTML. Furthermore, it must be pointed out that the extension deliberately violates some of the security guidelines for Visual Studio Code extensions, especially in the context of the usage of WebViews. These issues should probably be mitigated before using Rust Life Assistant in production.

Finally, it must also be pointed out that the Rust Life Assistant takes some assumptions about the system that it is running on. These are described in its "README.md"-file that is given in Appendix B.

5.6 Final assessment: Where the thesis's goals met?

The high-level goal of the thesis was to provide simple explanations for complex lifetime errors. We consider a lifetime error to be complex in the case when it is not fully explained by the Rust compiler error message, and an explanation to be simple when it provides all needed information (but no superfluous, confusing information) for understanding the lifetime error in an understandable fashion. Rust Life Assistant succeeds in doing so for a notable set of examples. Thereby we contribute a new way for (automatically) explaining lifetime errors.[18]

The goals also included getting more examples that feature lifetime error that can be analysed. We succeeded in getting more examples, and since some of them discovered limitations and issues in Rust Life (Assistant) they do cover some interesting cases. Unfortunately we were not able to categorize or classify these examples, and therefore we cannot really asses the test coverage that we achieve.[19] Another main goal was to create an improved graph output. We achieved this by acquiring one single path and by removing unnecessary intermediate lifetimes.

Furthermore, we improved the linkage form lifetimes to lines of source code both by fixing flaws in the previous work and by introducing a new approach for getting additional information. This linkage can be easily used by requesting highlighting of lines in an IDE extension, allowing the user to see the correspondence from graph nodes to lines of source code.

The overall goal of providing simple explanations for lifetime error was finally approached by generating a helpful textual explanation for the cause of a complex lifetime error. Anecdotally, after a demo of the tool a Rust user asked: "Why is the compiler not giving me these?" while looking at the points of text that Rust Life Assistant generated. As future work, the

[18] Actually, we ended up providing two different possible representations, but it is not clear how well suited that the graph-based approach is for being shown to end users.

[19] Doing so would be nice, but it was not exactly part of the original goals of this thesis.

generation of a similar step-by-step explanation could be added to the Rust compiler.

Chapter 6

Conclusion

As the evaluation shows, we were able to solve problems and contribute new approaches by this thesis. This is definitely an achievement while trying to provide better support to Rust programmers when dealing with lifetime errors. Still, there is more work to be done.

Once could start by applying some of the (simple) improvements that were indirectly proposed in the sections 5.4.1 and 5.5. By also providing a simple way for setting up Rust Life Assistant this might already be sufficient to get a first alpha/nightly prototype that could be released to the Rust community.

Additionally, we propose the following steps and ideas that could be worked on next.

- We recommend to apply the simple code improvements that were proposed in 5.5, especially those for Rust Life, as a first step before trying to implement new features of applying fixes for issues.
- In order to not get any superfluous lines in the Rust Life output it could help to not only search for one path in the `outlives` relation, but to search for the shortes one. Alternatively it could be worth to implement another step to optimize the graph that removes any circles in between variables. Thereby one might also want to recall that we never remove the first and the last node in the graph. It might be worth rethinking this decission.
- As mentioned, the parsing of source lines to get names of variables could probably be improved. Also, it might be worth to move this functionality from the Rust Life Assistant into Rust Life itself. This could also be helpful when implementing optimizations that operate on the level of variables.
- Additionally, it would probably be worth to further investigate the reason(s) for which Rust Life does not create any output for some

examples. (Both for some that contain loops, but also for those that contain error codes that we did not consider to be in our scope.)

- As indicated in previous chapters, the current Rust Life implementation does rather heavily relay on reading information from dump files that are created by the Rust compiler. It might be worth it to try to extract this information directly from the running compiler instance. This would resolve all issues that might exist in the process of reading and parsing this files. However, doing so would probably require to change some parts of the compiler so that it exposes the needed information as part of its internal API. Therefore, this would be a non-trivial endeavour that would take some time.
- Besides improving the code quality of Rust Life Assistant, it would also be beneficial to improve its visual appearance, e.g. by applying better styling to the textual representation. (However, before doing so it might be worth to restructure the code that generates the MTML and e.g. move the styling into an extra file.)
- Another option might be to implement the entire error explanation logic directly inside of the Rust compiler and then provide the text-based error explanation as part of a corresponding error message. This would provide easy accessibility for users, but for implementing it one would also need a sufficient knowledge of the Rust compiler's internals and possibly support form the core compiler developers. Another option might be available by using the compiler's plug-in API.[1]
- In the section 3.4, in the last paragraph other ways for visualizing lifetimes in Rust programs are mentioned. It might be worth further investigating some of these. (e.g. the usage of coloured lines next to the source code.) The output of Rust Life would probably allow to implement some of these as part of Rust Life Assistant.[2]
- One could conduct a survey across Rust users to get a better evaluation of the usability of the Rust Life Assistant, but also to find out more about their wishes and hopes regarding automated support for dealing with lifetime errors.

So even tough that this thesis reached some goals, there is more work in this area that could be done. Not all of it is probably suited for further research, but some of it does primarily require some engineering and implementation work.

[1] Using it was also considered for this thesis, but the idea was dropped since this API did not yet provide all possibilities and options that we considered to be necessary for building Rust Life.

[2] At least if it is possible to create the needed graphical elements by a Visual Studio Code extension.

Overall, the thesis demonstrates that in most cases of lifetime errors it is possible to generate step-by-step explanations that are easy to understand for the user. This technique could be implemented as part of the Rust compiler, for the benefit of both beginners and advanced Rust developers.

Appendix A

Naive Polonius borrowchecker rules, v0.8.0

There follows a list of all rules that are used by the Polonius borrow checker (naive version), that were found in the file `naive.rs`[1]. The rules are given as Datalog rules (using the Souffl project syntax), basically they are just copies of the comments in the source file. In addition, we also try to give short explanations of all rules, these explanations are mostly based on the information in the blog post by Nicholas Matsakis [3], i.e. basically the first introduction of the Polonius borrow checker.

This document gives the rules from version 0.4.0 of Polonius, i.e. the ones from Polonius-engine version 0.8.0.

Note that the construct that previously was called "loan", and often denoted as 'L' seems to now be called "borrow", and is now denoted as 'B'.

A.1 Rules

First, these are the rules that are used by Polonius, where the last rule is defining an actual (eventual) error.

The part of the Subset relation that is given as an input fact that is called outlives. This is not computed by the Polonius borrow checker, but given as static input.

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

Subset is transitive:

[1] polonius/polonius-engine/src/output/naive.rs

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

Propagates subset relationships across the control-flow graph edges:

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

Requires is described informally by the blog post by Nicholas Matsakis as follows:

> The region R requires the terms of the loan L to be enforced at the point P.
>
> Or, put another way:
>
> If the terms of the loan L are violated at the point P, then the region R is invalidated.

The first rule for requires says that the region for a borrow is always dependent on its corresponding loan:

```
requires(R, B, P) :-
  borrow_region(R, B, P).
```

The next rule says that if R1: R2, then R2 depends on any loans that R1 depends on:

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

This (basically) just propagates requires along cfg edges, but there is a twist. (The second-last line, giving !killed):

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

A loan L (the same as a borrow B) is live at the point P if some live region R requires it:

```
borrow_live_at(B, P) :-
  requires(R, B, P),
  region_live_at(R, P).
```

Finally, it is an error if a point P invalidates a loan L while the loan L is live:

```
.decl errors(B, P) :-
  invalidates(B, P),
  borrow_live_at(B, P).
```

A.2 Facts

As you might have noted, some of the relations that are used in the rules are not given by rules. As you might have guessed, these are so-called (input) facts. These are provided to Polonius as inputs from previous compilation phases. In principle, these are a representation of (parts) of the input program (input to the compiler), provided in a form that is well-suited for an analysis by Polonius and that especially points out the constraints that are implied by the input program and that must be satisfiable if the input shall be a valid Rust program.

This gives the outlive relationship, as the name already states. However, it actually seems to only give the part of the outlive relationshios that directly arise form the program code. The entire relationship is then defined by the `subset` rule. Therefore, this fact was called `base_subset` in the original blog post by Nicholas Matsakis.

```
.decl outlives(R1:Region, R2:Region, P:Point)
.input outlives
```

This fact is simply the control-flow graph (cfg) of the relevant program part, given as a set of edges that are connecting program points and thereby completely describe the graph.

```
.decl cfg_edge(P:Point, Q:Point)
.input cfg_edge
```

This basically gives the information that a certain region is live at a certain point. (The details of this, and the definitions are given in the blog

post, that also redirects to the NLL RFC.) However, note that in the current version of Polonius it does not directly use the `region_live_at` that is provided as input (as part of all_facts), but instad it uses a new one that is explicitly computed right at the beginning by Polonius by calling the method `liveness::init_region_live_at(...)`. (So the relation that is used by Polonius will differ from the one that is available as part of the inputs) Still, for the rules provided before, this (new) `region_live_at` is considered to be an input fact. Also, it seems (from the commit history) that this change was only introduced in Polonius-engine 0.8.0. Before, the `region_live_at` relation form the inputs was used directly.

```
.decl region_live_at(R:Region, P:Point)
.input region_live_at
```

Simply a quote of the description form the blog post by Nicholas Matsakis:

> This input is defined for each borrow expression (e.g., `&x` or `&mut v`) in the program. It relates the region from the borrow to the abstract loan that is created.

```
.decl borrow_region(R:Region, B:Borrow, P:Point)
.input borrow_region
```

Again, just a quote from the blog post by Nicholas Matsakis:

> `killed(L, P)` is defined when the point P is an assignment that overwrites one of the references whose referent was borrowed in the loan L.

For more details, pleas check the blog post that provided an illustrative example.

```
.decl killed(B:Borrow, P:Point)
.input killed
```

Finally, `invalidates` indicates that a Borrow will get invalid at a certain point. (Due to some operation, that is done at this point.)

```
.decl invalidates(B:Borrow, P:Point)
.input invalidates
```

Appendix B

README.md of Rust Life Assistant 0.0.2

This is an IDE extensions that does provides simple explanations for complex lifetime errors. For doing so it does use Rust Life.

B.1 Features

This extension can both show the graph-based (more precisely, path-based) visualization of an error that is created by Rust Life, as well as create a simple step-by-step guide in text form.

Both outputs are interactive, this means that parts of them (either the nodes of the graph or the blue parts of the text) can be clicked to request a highlighting of a related line of source code.

The graph-based visualization is requested by the command `Rust Life: Visualize as Graph`, and the text based by `Rust Life: Explain with Text` (Hit `Ctrl+Shift+P` to open the command plate.) Note that Rust Life might need some time to complete its execution, please be patient.

B.2 Requirements

This extension will only work on a system that is set up appropriately, since it has rather strict requirements. (It is not that flexible, esp. since some paths to files are hardcoded for now.)

Since no pre-built versions are available, one needs a working development environment for VS code extensions. This is best achieved by following the instructions in Your First Extension.

In addition, Rust must be installed. Since a specific nightly version is required we strongly recommend using rustup. (Pleas stick to default settings regarding paths for storing the files of rustup.)

Then, follow these steps to make everything ready for using Rust Life Assistant: - First you need to get a copy of the Rust Life executable. Currently, Rust Life Version 0.3.0 is required. Build it by following this steps: - cd to the `compiler_mod` directory. - run the command `make build` to start the build process. This might take some time. - Note that by doing so rustup will also install the `nightly-2019-05-21` toolchain, which must be installed to use Rust Life (Assistant). This will need approximately 1 GB of disk space. - The generated executable is located in `compiler_mod/target/debug` and called `extract-error` - Copy this executable into a folder named `.rust-life` in your home directory. (More precisely, in the home directory of the user that shall use Rust Life). Do not alter the name of the executable. - Now open the extension folder with the extension (`rust-life-assistant`) in VS code and hit F5 to build the extension and run it. - The extension development host (essentially another VS code instance) will start, and Rust Life Assistant is available for being used in it. - Open the Rust file that you want to analyse in it and run one of the commands that were described in the Section B.1.

Note: Due to the hardcoded file system paths this extension can only be used on GNU/Linux systems. It was only tested on an Ubunut-based platform, but will most likely also work on any other distributions that can run VS code and rustup. (However, it will most likely not work on Windows or macOS)

B.3 Known Issues

- If Rust Life fails, Rust Life Assistant might display old results from a previous execution. This is a simple implementation insufficiency that could be resolved by a simple clean-up step.
- In some cases the parsing of source lines to get local variable's names will fail and include something that is not the name of a local variable.
- Right now, there is not an official option to deactivate source code highlighting. Once it was triggered, one line will always stay highlighted. It can be deactivated by first closing the visualization that created it, then switching to a different tab (rendering the affected one invisible) and then switching back.
- Rust Life contains hardcoded paths to files, that probably prevent it from ruing on different platforms then GNU/Linux. Mitigating this would probably be doable, but might take some time.
- Some of the security guidelines for extensions, esp. for WebViews are currently violated. This should be fixed before using this extension in

production, esp. if it will eventually include using online content in the future.
- There probably are quite some more issues that we did not note yet or we forgot to include here.

Note: This list only provides issues specific to the implementation of Rust Life Assistant, some issues that mostly affect Rust Life are given in the thesis.

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*, ser. HILT '14. New York, NY, USA: ACM, 2014, pp. 103–104. [Online]. Available: http://doi.acm.org/10.1145/2663171.2663188

[2] 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

[3] N. D. Matsakis. (2018) An alias-based formulation of the borrow checker. Accessed on 2019/08/15. [Online]. Available: http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/

[4] (2018) Announcing Rust 1.31 and Rust 2018. Accessed on 2019/08/15. [Online]. Available: https://blog.rust-lang.org/2018/12/06/Rust-1.31-and-rust-2018.html#non-lexical-lifetimes

[5] Guide to Rustc Development. Accessed on 2019/08/23. [Online]. Available: https://rust-lang.github.io/rustc-guide/

[6] Crate rustc. Accessed on 2019/08/22. [Online]. Available: https://doc.rust-lang.org/nightly/nightly-rustc/rustc/index.html

[7] The MIR (Mid-level IR). Accessed on 2019/08/23. [Online]. Available: https://rust-lang.github.io/rustc-guide/mir/index.html

[8] N. Kay. (2019) Rust: The Hard Parts - Part One. Accessed on 2019/08/22. [Online]. Available: https://naftuli.wtf/2019/03/20/rust-the-hard-parts/

[9] N. D. Matsakis *et al.* (2017) NLL RFC. Accessed on 2019/08/22. [Online]. Available: https://github.com/nikomatsakis/nll-rfc/blob/master/0000-nonlexical-lifetimes.md#how-we-teach-this

[10] J. Walker. (2019) Rust Lifetime Visualization Ideas. Accessed on 2019/08/15. [Online]. Available: https://blog.adamant-lang.org/2019/rust-lifetime-visualization-ideas

[11] Borrow visualizer for the Rust Language Service. Accessed on 2019/08/15. [Online]. Available: https://internals.rust-lang.org/t/borrow-visualizer-for-the-rust-language-service

[12] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about Datalog (and never dared to ask)", *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 1, pp. 146–166, March 1989.

[13] Rust Compiler Error Index. Accessed on 2019/08/21. [Online]. Available: https://doc.rust-lang.org/error-index.html

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):

Simple Explanation of Complex Lifetime Errors in Rust

Authored by (in block letters):

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

Name(s):	**First name(s):**
BLaser	David

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, 23.08.2019	

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