Mechanical suggestions for Rust lifetime compiler errors

@mark-i-m on GitHub

Abstract

Rust is a new systems programming language that aims to allow low-level control and high performance, like C/C++, but with memory safety and data-race-freedom guaranteed statically. Rust accomplishes this using a compiler pass called the borrow checker. Unfortunately, the borrow checker's compile errors are sometimes difficult to understand. Beginners and advanced users alike may not be sure how to resolve them. In our work, we propose a tool that uses the compiler's analyses to mechanically produce suggestions for how to resolve lifetime-related errors. To evaluate the usefulness of our tool, we survey lifetime-related errors on Stack Overflow and the Rust Users Forum [?]. We find that our tool is helpful for generating fixes for 2 out of the 30 examples surveyed. For only one example, our tool produced a misleading suggestion. We also identify common areas of problems from the surveyed issue, including over-restrictive constraints, subtle interactions with advanced type system features, and fundamental design flaws in user's programs. We propose that these may be good areas to focus on improving compiler diagnostics.

1 Introduction

Rust is a systems programming language originally developed at Mozilla. The language aims to have performance comparable to C or C++ while at the same time guaranteeing memory-safety and data-race-freedom statically.

Rust accomplishes this primarily through its concepts of ownership and borrowing: each value in memory has exactly one variable binding that *owns* it. Other bindings may *borrow* the value by taking a reference. The compiler statically verifies that all access are safe.

Ownership and borrowing are key features of Rust that have led to its adoption for several major projects, including Dropbox [?], the Firefox web browser [?], and Facebook's experimental source control system Mononoke [?], to name a few.

The portion of the Rust compiler (rustc) that verifies these properties is called the *borrow checker.* The borrow checker verifies that values are initialized properly before use and that all references are well-formed [?].

Unfortunately, the borrow checker's error messages are notoriously hard to understand, despite significant work that has been put into improving them [?, ?]. Moreover, once one does understand them, a fix is not always self-evident. In some cases, only a simple change is need to fix the compile error; in other cases, the compile error indicates a fundamental bug in the program. In each Annual Rust Community Survey to date, lifetimes, borrowing, and ownership have consistently been ranked as "the most challenging concepts" of Rust [?, ?, ?].

In this work, we propose a tool for suggesting fixes to source code with borrow checker errors. We focus on outlives errors – those surrounding the relationships between the lifetimes of references – in our prototype. We also propose (but do not implement) extensions to our tool for some other types of lifetime errors.

Finally, we do a survey of borrow checker errors on Stack Overflow and the Rust Users Forum to evaluate our tool's usefulness. We find that our tool is helpful for generating fixes for 2 out of the 30 examples surveyed. For only one example, our tool produced a misleading suggestion. We also identify common areas of problems from the surveyed issue, including over-restrictive constraints, subtle interactions with advanced type system features, and fundamental design flaws in user's programs. We propose that these may be good areas to focus on improving compiler diagnostics.

2 Background

This section reviews key ideas in Rust, along with key algorithms of the borrow checker. We take advantage of these algorithms and ideas in our tool, which is implemented as modification to the compiler.

2.1 Ownership

In Rust, each *lvalue* (i.e. location in memory) is *owned* by at most one variable binding in the program. Assignment causes ownership to move to the new binding unless the type of the lvalue is copyable (i.e. can be safely bit-wise copied; for example, integer types are copyable).

Rust allows the creation of *references* to owned values. A reference can be *dereferenced* to get the lvalue, the *referent*. References can be *mutable* (*exclusive*) or *immutable* (*shared*). When a reference is taken to an lvalue, the reference is said to *borrow* the lvalue from the owner. The *lifetime* of a reference is the static portion of the program for which a reference is valid and can be dereferenced. Usually, the lifetime of a reference can be inferred by the compiler, but on occasion, the programmer may need to add annotations to the program to aid the compiler. In particular, such annotations can be used to restrict callers/users of a given function or type to prove certain properties about references used with the function or type (e.g. the caller must supply a reference whose lifetime outlives the contents of a given struct).

The compiler statically enforces certain properties about all lifetimes in a program to ensure memory-safety and data-race-freedom. The portion of the compiler that checks these invariants is called the *borrow checker*. Some examples of rules enforced by the borrow checker include:

- No dangling pointers: the referent of a reference must always be valid for the entire lifetime of the reference. This means that an lvalue should not be mutated during a borrow (including destruction or movement) except by the borrower through a mutable reference.
- No mutable aliasing pointers: the lifetime of a mutable reference cannot overlap with the lifetime of any other reference. That is, if a mutable reference exists to an lvalue, it must be the only reference (mutable or immutable) to that lvalue.
- Sharing: the lifetime of an immutable reference may overlap the lifetime of other immutable references. That is, there can be multiple immutable references to the same lvalue.

To check these rules (and others), the borrow checker must infer the lifetimes of all references and make sure they have or do not have certain relationships (e.g. one lifetime may be required to outlive another).

In the compiler, the terms *lifetime* and *region* are used interchangably.

2.2 Borrow Checking

Borrow checking happens near the end of compilation before code generation but after all other type checking passes. Borrow checking consists of three phases.

Move analysis The first phase does *move analysis*: computing which values are initialized and moved and where they move to [?, ?]. It also discovers illegal moves, such as moving out of statics. This pass is a traditional dataflow analysis.

Region Inference The second phase detects constraints on all regions and then infers the values of the regions themselves. There are two types of constraints: *liveness constraints*, which specify that a lifetime must be valid at a particular point in the control flow graph (e.g. because the reference is dereferenced there), and *outlives constraints*, which specify that one lifetime must last longer than another. Notationally, we will use the Rust syntax `'a` to refer to a lifetime named "a", and `'a: 'b` to denote "`'a` outlives `'b`".

Once constraints have been collected, they are used to infer the value of each region, which one can think of as a set of points in the control flow graph along with regions that must be outlived (which themselves will be sets). This is done by "propagating" constraints until a fixed point is reached: for all regions `'a` and `'b`, if `'a: 'b`, then all elements of `'b` are added to `'a`. In particular, if it is known that `'a: 'b`, then `'a` will contain a special element `rest('b)`, which denotes the rest of region `'b` [?].

Sanity Checking The third pass ensures that all relationships discovered during region inference are known to hold from the code. For example, if `'a` contains `rest('c)`, then it must be known from the code that `'a: 'c`. For example, this could hold because `'a == 'c` or because the user adds an annotation specifying that `'a` must outlive `'c`. If a relation is not known to hold, then a compile error is reported [?].

For this project, there are two special cases handled by the compiler which we will ignore for simplicity. First, Rust also allows more advanced types of lifetime constraints known as *higher-ranked trait bounds*, but they are quite subtle and less common in programs. Second, Rust has closures, which may close over a reference in its environment. In such a case, the constraints are propagated to the environment. We do not account for such propagation in out tool for simplicity.

3 Mechanical Fixes

In this section, we propose a way to produce mechanical suggestions for compile errors. We focus on two simple classes of borrow checker errors: missing outlives constraints and move errors. In the following sections, we describe our implementation and evaluation of the proposal for outlives constraints. To simplify discussion, we begin with simple examples.

3.1 Outlives constraints

Errors about outlives constraints result when region inference discovers that `'a: 'b`, but the user's code does not include such a restriction.

3.1.1 Examples

Listing 2: Does Not Compile

```
fn foo<'a, 'b>(
    x: &'a usize // parameters
) -> &'b usize {
    x
}
```

Some notes on rust syntax:

- The `<'a, 'b>` declares type parameters for the function `foo`. In this case, we only have the two lifetime parameters `'a` and `'b`, but in general we may also have other generics.
- `&'a usize` denotes a reference with lifetime `'a` to a machine-width unsigned integer (64-bits on my machine).
- `-> &'b usize` denotes that `foo` returns a reference with a lifetime `'b` to a machine-width unsigned interger.

Because we know nothing about `'a` or `'b` with respect to each other (from the code), it would be unsafe to allow `foo` to return `x` because we cannot prove that it will live for lifetime `'b`. If we allowed this code, the returned reference may outlive the input reference and could become a dangling reference.

Listing **??** shows the compiler error for Listing **??**. As the error message suggests, this error can be resolved by telling the compiler to check that no caller should be allowed to hold on to the returned reference for longer than it holds on to the input reference. We do this with a `where` clause:

Listing 3: Fixed

```
fn foo<'a, 'b>(
    x: &'a usize
) -> &'b usize
where
    'a: 'b,
{
    x
}
```

Alternately, we can indicate that the input and output have the same lifetime, which happens to be true in this case.

Listing 4: Alternate Fix

```
fn foo<'a>(
    x: &'a usize
) -> &'a usize {
    x
}
```

In Rust, this is a common case, and the lifetimes can just be elided altogether:

Listing 5: Alternate Fix

```
fn foo(x: &usize) -> &usize { x }
```

Appendix A has more examples of outlives errors and fixes.

3.1.2 Fixing

At a high level, we propose the following simple fix for outlives errors: if the lifetime error involves implicit lifetime parameters (e.g. Listing **??**), make them explicit. Then, add any outlives constraints needed.

More specifically, during the "sanity checking" phase, when the borrow checker discovers an unsatisfied outlives constraint `'a: 'b`, we check if the error involves a region in the function header. If it does, we add the outlives constraint to a set of unsatisfied constraints to be added to the suggestion.

In order to produce more concise suggestions, we collect all outlives constraints for the same lifetime: `'a: 'b, 'a: 'c` can be written as `'a: 'b + 'c`. This is done by simply waiting until the end of sanity checking before producing any error messages so that we know we have seen all constraints that must be added.

Finally, because lifetimes are often not explicit in the source code, we may need to synthesize a new name for region to use in the suggestion. The suggestions in our prototype assume that lifetime names of the form `'generatedlN`, for integers N, are unused

Listing 1: Compiler Error for Listing **??**

```
error: unsatisfied lifetime constraints
 --> samples/s2.rs:7:5
  |
3 | fn foo<'a, 'b>(x: &'a usize) -> &'b usize
  |        --  -- lifetime ''b' defined here
  |        |
  |        lifetime ''a' defined here
...
7 |     x
  |     ^ returning this value requires that ''a' must outlive ''b'
```

by users, but a production-quality implementation would leverage the compiler's knowledge of in-scope names to choose a truely unique (and more idiomatic) name.

3.1.3 Correctness

The suggestion scheme of the previous subsection produces minimal suggestions. That is, they add the least-restrictive outlives constraints possible to produce a fix.

This actually follows by construction of the borrow checker, which transitively computes which lifetime relationships must be true given the control flow and compares against what is known to be true (e.g. from `where` clauses). Specifically, given the value of an inferred region `'a` (recall that this is a set), if `rest('b)` is in `'a`, the source code must declare or imply such a constraint on callers. The lack of such a constraint is, by definition, an error, so adding such a constraint is, by construction, correct.

Note that, adding outlives constraints *changes* the semantics of the program by restricting callers of the function. In fact, any such suggestion must change the semantics of the program because the current semantics represent a compile error. Thus, it's worth noting that at best, the "fixed" program represents a reasonable suggestion for what the user might mean, but we cannot know for sure without more info.

Importantly, adding lifetime constraints does *not* change the *behavior* of a function; rather, it restricts the set of possible contexts in which a function can be called. This means that the suggestions made by the suggestion scheme described above can be made more confidently than most suggestions: we cannot introduce a bug that was *not already there*. Notice, though, that our suggestion may exacerbate a problem that is already there in some cases.

3.1.4 Potential Future Extensions

Another potentially useful fix is to remove over-restrictive outlives constraints from code. Given the results of region inference, we may be able to remove an outlives constraint `'a: 'b` from source code if the inferred regions corresponding to `'a` and `'b` do not have `rest('b)` $\in$ `'a`. We do not explore this idea further in this work.

3.2 Move errors

In Rust, values move by default. That is, an assignment changes *ownership* of a value to a new variable binding. Some types (e.g. integers) are copyable, which means that if a move would cause a move error, we make a copy of the value instead. In this work, we focus on non-copyable types for simplicity.

We focus on one type of move error in particular: a value is used after it is moved. This type of error can occur in many different contexts. For simplicity, we focus on move errors that don't involve subpaths (e.g. fields of structs). However, we surmise that our proposal extends to subpaths neatly.

3.2.1 Examples

Listing 6: Does Not Compile

```
struct Foo; // not copyable

fn move_it(_foo: Foo) {}

fn use_it(_foo: &Foo) {}

fn main() {
    let foo = Foo;

    move_it(foo);
    use_it(&foo);
}
```

In this example, we define a new empty struct `Foo`, which is not copyable. We define two functions: `move_it` takes a `Foo` by value, meaning that the argument *moves* into the parameter `_foo`; and function `use_it` only *borrows* its argument. In `main`, because the `Foo` lvalue has moved out of `foo` and into `move_it`, `use_it` cannot borrow the value, resulting in the error shown in Listing **??**.

There are a few possible fixes one could apply. Unlike the lifetime errors, it is less obvious which option more closely matches the intentions of the author of the code. The first option is to reorder the calls so that `foo` is borrowed and then moved afterwards.

Listing 8: Fixed

```
struct Foo;

fn move_it(_foo: Foo) {}

fn use_it(_foo: &Foo) {}

fn main() {
    let foo = Foo;

    use_it(&foo);
    move_it(foo); // reordered
}
```

It is also possible that the author did not intend for `move_it` to take its parameter by value (e.g. as the result of a refactor).

Listing 9: Fixed

```
struct Foo;

// now takes a reference
fn move_it(_foo: &Foo) {}

fn use_it(_foo: &Foo) {}

fn main() {
    let foo = Foo;

    move_it(&foo); // `foo` not moved
    use_it(&foo);
}
```

3.2.2 Fixing

There seem to be two options for possible fixes.

1. Reorder the move site and borrow site; the lvalue should be borrowed before it is moved. Clearly, this could change program behavior. However, in some cases, the two lines in question (and any in-between) are commutative, so this is sometimes a reasonable fix. This fix is represented by Listing **??**.

2. Change the move site to a borrow site so that the value can later be borrowed again. This is feasible only if no other code uses the fact that the value moved. For example, in Listing **??**, `move_it` does not explicitly use the ownership of its parameter value in any way. One caveat is that in some cases, an API may take an argument by value intentionally to prevent it from being used again, so this fix is not always correct.

We do not implement either of these strategies in our prototype. However, both seem implementable:

The first strategy can be implemented based on the move analysis results. The move analysis computes each place where a value is moved. The suggestion can then reorder all borrows before the first move.

The second strategy can be implemented by checking the scope of a binding that receives ownership of a moved lvalue (e.g. a function body). If this scope does not require ownership (i.e. the value does not move again), then we can change the move to a borrow.

3.2.3 Correctness

Unlike the proposal in Section **??** for lifetime errors, our proposals in Section **??** for move errors are strictly heuristic – based entirely on the experience of the author. Unlike adding lifetime constraints, changing the order of statements or the type of arguments does change the runtime behavior of code. Thus, proposing one of the suggestions above may actually introduce a bug that was not previously there, unlike our lifetime error suggestion scheme.

4 Implementation

We implement a prototype of the proposed suggestion scheme of Section **??** for outlives errors. Our prototype is implemented as a modified version of the compiler. Our modifications consist of 885 changed lines of Rust in the compiler (749 additions and 136 deletions). The implementation can be found at `https://github.com/mark-i-m/rust/tree/synthesis`.

Our implementation borrows much of the existing machinery already in place for borrow checking and

Listing 7: Compiler Error for Listing **??**

```
error[E0382]: use of moved value: `foo`
  --> m1.rs:11:13
   |
10 |     move_it(foo);
   |             --- value moved here
11 |     use_it(&foo);
   |             ^^^ value used here after move
   |
   = note: move occurs because `foo` has type `Foo`,
           which does not implement the `Copy` trait

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
```

error reporting in the compiler. The implementation collects missing outlives constraints from the results of region inference and sanity checking. It then uses the compiler's own diagnostics and error reporting machinery to output a formatted suggestion message similar to those output by the compiler for other errors.

One change made from the compiler's current algorithm is that the compiler stops checking for errors once it finds the first error to avoid overwhelming the user with errors. However, for our suggestions, we would like to find all errors so that we can output a suggestion that covers all of them in one suggestion. Thus, we do not stop after the first error is detected.

Limitations As mentioned previously, our implementation does not handle lifetime errors involving closures or higher-ranked trait bounds. Another limitation is that our implementation assumes that lifetime constraints are missing on functions or methods, rather than on type definitions (e.g. structs, `impl` blocks, etc).

In the following section, we evaluate how useful our tool is despite these limitations.

5 Evaluation

To evaluate the usefulness of our tool, we do a survey of borrow-checker related questions on Stack Overflow and the Rust Users Forum. Appendix B lists the issues we examined.

Selection Criteria To select issues from Stack Overflow, we searched for issues tagged with "rust" and "lifetime" using the keyword "error". We sorted the results in order of decreasing "relevance" (the default ranking). Questions that do not involve an error (e.g. questions about Rust design choices) were omitted from the survey.

To select issues from users.rust-lang.org, the Rust Users Forum, we searched for issues categorized as "help" using the search term "lifetime error", sorted in decreasing order of "relevance" (the default ranking). Errors stemming from a poster's ignorance of Rust syntax were skipped.

We choose the top 15 eligible results from Stack Overflow and users.rust-lang.org (30 total). Issues which could not be reproduced were omitted. We believe this methodology gives a reasonable sampling of common errors.

For each issue, we extract the minimum reproducible example from the original poster or top response, whichever comes first. We then compile the example with our tool (a modified compiler). If a suggestion is produced, we apply the suggestion to see if it resolves the compile error.

5.1 Results

Appendix B lists the issues that were surveyed and whether our tool produces a useful suggestion for each issue.

We found that posts on Stack Overflow often dealt with advanced but fundamental issues in Rust. For example, one common problem was returning a reference to a local variable on the stack. This is always a bug since the stack frame becomes undefined when the function returns. This is a common bug in C and C++, too.

In contrast, we found that posts on users.rust-lang.org often dealt with subtle or complex problems.

Often, they had to do with advanced type system interactions or edge cases in type inference. Also, many of these issues involved code from real projects, rather than contrived or distilled reproductions of errors.

In this section, we will refer to issue numbers from the tables in Appendix B. An issue SO.N refers to Stack Overflow issue N (the Nth entry in the Figure **??** of Appendix B), and URLO.N refers to the Nth Users Forum issue (Figure **??** of Appendix B).

Our tool was able to produce suggestions for two issues (SO.1 and SO.15) out of the surveyed 30 issues. In only one case (URLO.15), we produce a misleading suggestion.

The rest of this section discusses a few interesting failures of the tool. We then discuss idiomatic suggestions. Finally, we discuss common problems observed in the survey.

5.1.1 Interesting failures

SO.12 In Rust, abstraction is often done via *traits*. A trait is a named collection of methods that may be implemented for a type. Functions can *bound* generic type parameters by requiring that they be any type that implements a given trait. When a type implements a trait, it's implementation must exactly match the method header definitions in the trait definition, including any bounds on type parameters and lifetimes.

In SO.12, our tool does indeed produce a valid suggestion. However, applying the suggestion causes the a trait implementation to mismatch with the trait definition. Updating the definition to also use the suggestion resolves the compile error.

This represents an implementation limitation of our tool. In principle, the tool could check if a suggestion should also update a trait. One caveat is that the trait may be defined in another library such that the suggestion cannot be applied to it.

URLO.4 In this example, the user's code is fundamentally problematic: it uses advanced type system features to produce a type that the type system effectively views as a self-referential struct (a struct where one field is a pointer to another field in the same struct). Self-referential structs are impossible in Rust (for now) because moving them causes all pointers into the struct to dangle, breaking the memory safety guarantee that all references are valid.

URLO.4 represents a (rare) case where a fundamentally buggy program receives a bizzare and seemingly unrelated error. To make matters worse, the user's code also contains an incorrect lifetime bound. Removing this bound produces an outlives error, which our tool produces a suggestion for. This is an example of code where our tool produces a misleading suggestion that does not fix an underlying and more fundamental problem. Even human experts on the Forum were unable to give confident advice on this issue, so it is not clear if a mechanical suggestion is possible. Out of all 30 surveyed issues, this was, in the author's opinion, the only case of a misleading compile error caused by the confluence of many problems along with use of advanced type system features.

URLO.15 This is the only example where our tool gave a suggestion on unmodified code that was misleading. The fundamental problem is an over-restrictive bound placed on a trait definition. This leads our tool to believe that there are unsatisfied outlives constraints elsewhere in the program, which it suggests adding. Adding these outlives constraints causes the error to move elsewhere. The correct solution is to remove the lifetime bounds from the trait. This issue demonstrates a key limitation of our tool: it is unable to reason about advance interactions of lifetimes with other type system features, particularly higher kinds such as traits.

5.1.2 Idiomatic code

In Rust, idiomatic code is almost always what the programmer intends to write. Unfortunately, many of the suggestions produced by our tool are unidiomatic.

In particular, it is unidiomatic to name the lifetime of `&self`, a method receiver. The compiler almost always infers the right lifetime for it, and adding an explicit lifetime is usually needlessly restrictive. (SO.1 represents a rare case where this is not true).

However, we found that due to our implementation, our tool often added such bounds. For example, consider SO.15 (Listing **??**). Figure **??** shows the suggestion produced by our tool. In this example, the `F::get` method has an unmet outlives constraint: `'b: 'a`. The suggestion of our tool is to add this constraint, but a better solution would be to elide both lifetimes `'a` and `'b` because the compiler infers the right lifetimes in this case.

Listing 11: SO.15

```
struct F<'a> {
    x: &'a mut i32,
}
impl<'a> F<'a> {
    fn get<'b>(&'b self) -> &'a i32 {
        self.x
    }
```

Listing 10: Fixed

```
help: the following suggestions may resolve your lifetime errors
help: add the following outlives constraint(s)
  |
5 |     fn get<'b>(&'b self) -> &'a i32  where
6 |       'b: 'a,
  |
```

```
}
```

5.1.3 Classification of other failures

Of the remaining surveyed issues, we observe three broad (not mutually exclusive) bands of common problems reported by users.

The first involves issues that simply cannot be solved by adding an outlives constraint. In particular, common problems include over-restrictive constraints such as naming a lifetime that doesn't need to be named and/or coupling it unnecessarily to another lifetime (SO.9, URLO.5, URLO.10, URLO.14, URLO.15). Section **??** proposes some ideas for how to make suggestions for some of these problems.

The second band involves surprising, subtle, and/or weird interactions with other type system features such as inference defaults (URLO.1, URLO.10), traits and trait objects (SO.13, SO.14, URLO.10, URLO.13, URLO.15), generics (URLO.3), subtyping variance (SO.6, URLO.6), or closures (URLO.8). These errors seem like very difficult errors for which to try to produce suggestions, but they may be high value targets because they tend to be extremely confusing.

The third band involves programs that are incorrect or impossible in Rust. This usually means that either the user is doing something fundamentally wrong, or that what the user is doing is not easy to express in Rust for safety reasons. Our survey primarily found the following issues: self-referential structs and iterators (SO.2, URLO.4), returning references to locals on the stack (SO.3, SO.5, SO.7, SO.8, URLO.11), and limitations of the standard threading API (SO.4, SO.10, URLO.9). For errors about self-referential types and the threading API, the user's code likely has a design flaw, so emitting a suggestion seems futile. Rather, error messages could be improved to explain why the user is getting the error. For errors in which the user returns a reference to a stack variable, we may be able to produce suggestions in which the user returns an owned value, rather than a reference.

While we haven't investigated these areas, one may surmise that the strongly-typed nature of Rust makes it possible to synthesize useful suggestions in many of these other errors as well.

5.2 Discussion

Our tool makes useful suggestions for 2 out of 30 of the surveyed issues. This demonstrates that mechanically-produced suggestions are possible and can be useful. Nonetheless, significant effort needs to be put into implementation to handle edge cases, as in the case of SO.12, and to make suggestions use the simplest idiomatic Rust possible.

Moreover, our survey demonstrates that targetted improvements to compiler error messages and/or producing suggestions may have high impact:

- Suggest removal of unnecessary annotations;
- Improve error messages for errors that result from surprising defaults or advanced type system features such as subtyping variance;
- Improve error messages for errors resulting from interactions with traits, trait objects, and higher-ranked trait bounds; possibly also try to suggest fixes where feasible;
- Improve error messages about self-referential structs;
- Improve error messages about using references stack variables that have moved or are not initialized any more;
- Improve error messages around the standard threading API and closures

6 Conclusion

We describe ways to use the compiler's internal analyses to generate mechanical, useful suggestions for resolving Rust compile errors involving the borrow checker. We propose schemes for generating suggestions for lifetime outlives errors, move errors, and other lifetime errors.

We evaluate our suggestion scheme for lifetime outlives errors by surveying issues on Stack Overflow and the Rust Users Forum. We find that while our tool demonstrates potential by producing useful fixes for 2 out of 30 of the surveyed issues, much work is needed to make a production-quality useful tool.

We also identify common areas of problems from the surveyed issue, including over-restrictive constraints, subtle interactions with advanced type system features, and fundamental design flaws in user's programs. We suggest that these may be good areas to focus on improving compiler diagnostics.

Appendix A: More examples of errors and fixes

Here are a few more examples and possible fixes:

Listing 12: Does Not Compile

```
fn foo<'a, 'b, 'c>(
    x: &'a usize
) -> (&'b usize, &'c usize) {
    (x, x) // tuple of references
}
```

Listing 13: Fixed Listing **??**

```
fn foo<'a, 'b, 'c>(
    x: &'a usize
) -> (&'b usize, &'c usize)
where
    // 'a outlives 'b and 'c
    'a: 'b + 'c,
{
    (x, x)
}
```

In this case, since the output depends only on the input, it is valid to change all lifetimes to `'a`.

Here is a similar example:

Listing 14: Does Not Compile

```
fn foo<'a, 'b>(
    x: &'a usize,
    y: &'b usize,
) -> &'a usize {
    y
}
```

Listing 15: Fixed Listing **??**

```
fn foo<'a, 'b>(
    x: &'a usize,
    y: &'b usize,
) -> &'a usize
where
    'b: 'a,
{
    y
}
```

Finally, here is a more complicated example taken from Stack Overflow (SO.1).

Listing 16: Does Not Compile

```
use std::cell::Cell;

struct Bar<'a> {
    bar: &'a str,
}
impl<'a> Bar<'a> {
    fn new(foo: &'a Foo<'a>) -> Bar<'a> {
        Bar { bar: foo.raw }
    }
}

pub struct Foo<'a> {
    raw: &'a str,
    cell: Cell<&'a str>,
}
impl<'a> Foo<'a> {
    fn get_bar(&self) -> Bar {
        Bar::new(&self)
    }
}
```

In this example, we define a struct `Bar`, which has the lifetime type parameter `'a` and has one field (a string slice) with that lifetime. `Bar` also has one method `new`.

We also define a struct `Foo`, which has two fields: a reference `raw` (string slice) and a `Cell`. The `get_bar` function is a method on `Foo`. It takes `&self` (the method receiver) by reference. In this case, the author chose not to annotate `&self` or `Bar` with an explicit lifetime (this is actually the common case in rust).

Without getting into the details, the significance of `Cell` is that it is invariant (as in "not covariant or contravariant") over it's type parameters (`'a`). This means that the compiler will not try to infer a subtyping (outlives) relationship between the lifetime of

&self and Bar. This implies that we cannot prove that the Foo object will outlast the Bar object, which would be memory unsafe.

We can fix this error, by explicitly telling the compiler that the lifetime of self outlives the lifetime of any reference in the return value. As before, we could also use the same lifetime, since the output depends only on the input.

Listing 17: Fixed

```
impl<'a> Foo<'a> {
    fn get_bar<'b>(&'a self) -> Bar<'b>
    where
        'a: 'b,
    {
        Bar::new(&self)
    }
}
```

This is one case where our tool works despite interactions with variance.

Appendix B: Survey of Stack Overflow and users.rust-lang.org

This section lists the issues surveyed for the Evaluation of our prototype. The "Fixed" column indicates whether the suggestion scheme proposed in Section **??** for lifetime errors solves the reported error. The issues for the results presented in this paper were collected from December 6, 2018 through December 8, 2018, though we had previously done unsystematic searches through Stack Overflow and the Users Forum to build our own intuition.

Figure 1: Surveyed issues from Stack Overflow

	Issue	Fixed	Link
1	Incorrect/Missing lifetime constraint	Yes	`https://stackoverflow.com/questions/41204134/rust-lifetime-error`
2	Self-referential structs	No	`https://stackoverflow.com/questions/32300132/why-cant-i-store-a-value-and-a-reference-to-that-value-in-the-same-struct`
3	Returning reference to stack local	No	`https://stackoverflow.com/questions/30421603/implementing-error-description-with-own-string`
4	Lifetimes across threads	No	`https://stackoverflow.com/questions/27619047/lifetime-error-when-spawning-a-task`
5	Storing reference to local that is dropped	No	`https://stackoverflow.com/questions/36949184/error-line-does-not-live-long-enough-but-its-ok-in-playground`
6	Subtyping variance	No	`https://stackoverflow.com/questions/50566942/adding-unrelated-generic-parameter-triggers-strange-lifetime-error`
7	Returning reference to stack local	No	`https://stackoverflow.com/questions/41529200/rust-lifetime-variable-does-not-live-long-enough-error`
8	Returning reference to stack local	No	`https://stackoverflow.com/questions/32682876/is-there-any-way-to-return-a-reference-to-a-variable-created-in-a-function`
9	Lifetimes need to be decoupled	No	`https://stackoverflow.com/questions/33205952/how-to-solve-this-lifetime-related-error`
10	Lifetimes across threads	No	`https://stackoverflow.com/questions/28154563/threading-lifetime-error`
11	Lifetimes of mutable borrows overlap	No	`https://stackoverflow.com/questions/33727808/lifetime-errors-using-filter-map`
12	Incorrect/Missing lifetime constraint	*	`https://stackoverflow.com/questions/43454783/error-e0495-when-returning-an-element-from-a-vector`
13	Missing outlives constraint to match trait definition	No	`https://stackoverflow.com/questions/24847331/rust-lifetime-error-expected-concrete-lifetime-but-found-bound-lifetime`
14	Missing lifetime bound on trait object	No	`https://stackoverflow.com/questions/39339419/confusing-error-in-rust-with-trait-object-lifetime`
15	Incorrect/Missing lifetime constraint	Yes	`https://stackoverflow.com/questions/41417057/why-do-i-get-a-lifetime-error-when-i-use-a-mutable-reference-in-a-struct-instead`

* The prototype does produce a suggestion for this example, and the suggestion does resolve lifetime errors. However, the suggestion in it's raw form does not compile because adding an outlives constraint causes the implementation to mismatch with a trait definition. If the definition is updated to also use the suggested fix, the example compiles.

Figure 2: Surveyed issues from users.rust-lang.org

	Issue	Fixed	Link
1	Needs explicit lifetime	No	`https://users.rust-lang.org/t/trait-alias-causing-lifetime-error/16131`
2	Need lifetime bound on struct	No	`https://users.rust-lang.org/t/lifetime-error-for-containered-return-type/11542`
3	Unknown lifetime of generic type parameter	No	`https://users.rust-lang.org/t/piston-and-textures-lifetime-error/12613`
4	Incorrect lifetime bounds with generics and self-referential structs	**	`https://users.rust-lang.org/t/solved-object-function-call-causes-lifetime-error-i-do-not-understand/12401`
5	Over-restrictive lifetime constraint	No	`https://users.rust-lang.org/t/having-compiling-due-to-lifetime-error/3536/3`
6	Missing lifetime constraint that interacts with subtyping variance	No	`https://users.rust-lang.org/t/workaround-found-cannot-infer-an-appropriate-lifetime-error-borrowing-from-a-mutable-field/5103`
7	Limitations of reborrows for nested mutable references	No	`https://users.rust-lang.org/t/weird-borrowck-issues-with-mut-mut-u8/13852`
8	Incorrect lifetimes of closure arguments	No	`https://users.rust-lang.org/t/lifetime-error-with-iterators-and-closures/8782/4`
9	Lifetimes across threads	No	`https://users.rust-lang.org/t/sharing-a-cache-of-non-static-references-across-threads/7617`
10	Over-restrictive lifetime constraint	No	`https://users.rust-lang.org/t/lifetime-error-with-trait-object/20113`
11	Returning reference to stack local	No	`https://users.rust-lang.org/t/solved-string-lifetime-error-borrowed-value-does-not-live-long-enough/5233`
12	Missing lifetime in procedural macro invocation	No	`https://users.rust-lang.org/t/lifetime-error-of-serde-s-impl-deserializer-example/19092/3`
13	Higher-ranked trait bound lifetime constraint and weird inference rule	No	`https://users.rust-lang.org/t/pls-help-to-explain-weird-error-with-closure-lifetime-inference/22843`
14	Over-restrictive lifetime constraint	No	`https://users.rust-lang.org/t/lifetime-do-not-understand-error-message/15822`
15	Over-restrictive lifetime constraint on trait	***	`https://users.rust-lang.org/t/solved-cannot-infer-lifetime-vs-borrowing-more-than-once-errors/2671`

** In this example, the user is trying to use the type system to produce a safe allocator API where pointers can never outlive the allocator. However, they have incorrect bounds that make it impossible to use the API. Removing the incorrect bound also leads to compile error; however, the suggestion produced by our tool for the new compile error recommends putting back the bound, which leads to the original borrow checker error in another function. This is due to a fundamental bug in the program.

*** In this example, the user adds an unnecessary bound on trait definition. This leads our tool to believe that a constraint must be added to the trait implementation. However, adding the bound causes different errors. The correct solution is to remove the bound from the trait.