A formalism to reason about the safety of lifetime annotations on function signatures in Rust

TLC Project Report

Vikram Nitin - vn2288

May 13, 2023

1 Introduction

The Rust programming language is unique for the memory safety guarantees that it offers. This is thanks to a sophisticated system of ownership, borrowing, and borrow checking. We shall first briefly discuss those concepts, followed by a longer discussion about lifetimes and lifetime annotations.

1.1 Ownership

Each variable in Rust owns the memory associated with the variable. Whenever a variable goes out of scope, the compiler frees the associated memory—this feature provides protection against memory leaks. Each resource/memory region can have only one owner.

1.2 Borrowing

Variables can be accessed by borrowing, analogous to a reference in C. Depending on whether the borrower can modify the borrowed value or not, borrows can be mutable (`&mut`) or immutable (`&`), respectively. Rust has a Borrow Checker that decides at compile time whether each borrow to a variable is permitted or not.

1.3 Lifetimes

Rust uses lifetimes, a construct that assists the borrow checker in verifying the validity of each borrow. Lifetimes are similar to scopes, but subtly different[1]

[1] `http://smallcultfollowing.com/babysteps/blog/2016/04/27/non-lexical-lifetimes-introduction/`

The borrow checker ensures the following two rules :

Rule 1: *The lifetime of a borrow cannot be longer than the lifetime of the borrowed value.* Such a rule ensures that memory safety issues like use-after-free or dangling pointers cannot happen as it disallows any references that outlive the owner.
Rule 2: *In a given scope, for a given variable, one can either have a single mutable borrow or any number of immutable borrows.* This is also implemented using lifetimes - the compiler checks if the lifetimes of the borrows overlap.

Lifetime Annotations: Within a function, the compiler infers all lifetimes automatically. But once a variable *crosses function boundaries*, in some cases the compiler does not automatically assign a lifetime. Consider a function that takes two borrows as inputs and returns one borrow.

```
fn foo(x: &String, y: &String) -> &String {/*..*/}
```

The returned borrow must use one of the two input borrows, but there is ambiguity about which one. The two input borrows could have different lifetimes, in which case an appropriate lifetime must be assigned to the returned borrow. Now although the compiler could technically look at the body of the function and infer the lifetime of the returned borrow, Rust makes a deliberate choice *not* to infer certain borrow lifetimes in function signatures[2]. In this case, the lifetimes must be specified using *lifetime annotations*.

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

Here `'a` and `'b` are lifetime parameters. This function signature indicates that the function takes two borrows, and returns another borrow with the lifetime of the *second* input borrow. The implementation, i.e., the body of the function, must match its signature.

At compile time, the compiler assigns concrete lifetimes to each lifetime parameter.[3] If a borrow is annotated with lifetime parameter `'a`, then the borrowed value remains borrowed for the *entire* concrete lifetime associated with `'a`. In conjunction with Rule 1, this gives us the following interpretation of lifetime annotations.
Rule 3: *If a borrow is annotated with lifetime parameter `'a`, then the borrowed value must live longer than (must outlive) the concrete lifetime corresponding to `'a`.*
Throughout the rest of this report, we shall use "a borrow has lifetime `'a`" as shorthand for "a borrow is annotated with lifetime parameter `'a`".
Structures can also have lifetime annotations. If a structure `Foo` contains a borrow with lifetime `'a`, then the structure must be annotated with the same lifetime, like

[2]Except in some specific cases. See `https://doc.rust-lang.org/nomicon/lifetime-elision.html`

[3]A recent reformulation of the borrow checker conceptualizes borrows as sets of *loans*, not regions.

`Foo<'a>`. This means that all the "underlying" values of the structure outlive `'a`. And if a structure has no lifetime annotations, then it contains only owned values and no borrows.

1.4 Raw Pointers and Unsafe Rust

Raw pointers (`*const`, `mut`) are similar to pointers in C - they obey none of the guarantees that borrows are expected to obey. For example, one can have mutable and immutable raw pointers to the same location. The values they point to need not live as long as the pointer itself, which means that they are not guaranteed to point to valid memory and can even be null. They also don't have lifetime annotations. Although *creating* a raw pointer is allowed in normal (safe) Rust, *dereferencing* a raw pointer requires "`unsafe`" code.

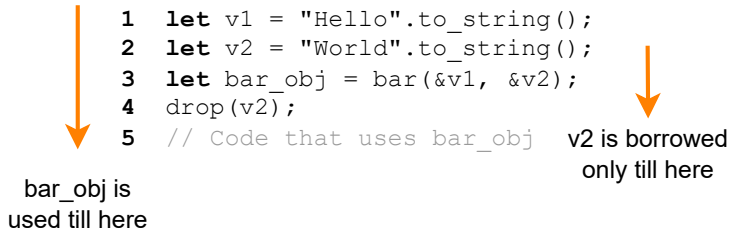
1.5 Unsafe Code and Lifetime Annotations

Although raw pointers themselves don't have lifetime annotations, they can be part of an Algebraic Data Type (like a struct) with associated lifetime annotations. In Figure 1a, we return a structure containing a raw pointer, and the structure is annotated with a lifetime of the input borrow `x`.

The key difference here is that Rust *cannot* automatically infer the lifetimes in the function signature, because it does not track lifetimes through raw pointers. So it relies on our provided annotations in the signature being compatible with the implementation of the function. In Figure 1a, the signature now indicates that the returned structure object has the lifetime of `x`, but the returned structure contains a pointer to `y`. As shown in Figure 1b, this could cause a use-after-free error, which is an example of Undefined Behavior (UB). In this project, I attempt to characterize such use-after-free bugs occurring due to incorrect lifetime annotations on function signatures.

```
struct Foo<'a> {
    inner: *const String,
    ..
}
fn bar<'a,'b>(x: &'a String, y: &'b String) -> Foo<'a> {
    Foo{inner: y as *const String, ..}
}
```

(a) The function returns a pointer to the first borrow, but the annotation indicates that it uses the second borrow.



Code compiles, but potential memory safety error!

(b) The string `v2` is dropped on Line 4, and `bar_obj` will contain a pointer to freed memory. This can cause a use-after-free error.

Figure 1: Incorrect lifetime annotations on functions can cause memory safety errors.

2 Characterizing Lifetime Annotation Errors

At any point in the program, if there is an active borrow or raw pointer that points to de-allocated memory, then this is potentially dangerous and could cause a "use-after-free error" when dereferenced. Let us focus on **use-after-free** errors for the rest of this report.

Recall the meaning of lifetime annotations:

- If a borrow has lifetime annotation `'a`, then the borrowed value lives for at least `'a`.
- If a structure has a lifetime annotation `<'a>`, then it contains a borrow annotated with lifetime `'a`.

We are concerned with function signatures and the types appearing in them. We notice that within a function, some values may be transferred between two arguments or between an argument and returned value. When this transfer of values happens, the lifetime annotations on the *target* of the transfer should be consistent with the transferred value. This is analogous to taking an object out of one container and placing it in another container - we want to make sure that the "label" on the second container is consistent with what is being placed in it.

The goal of my report is to come up with a set of rules to check if the "labels" on returned types are consistent with what we know about the values being transferred to them from function arguments. In contrast to other works attempting to formalize some subset of Rust, I do not try to reason about the statements in the function body.

Key Assumption: We assume that some all-knowing oracle exists that has told us that there is a value that has been transferred from a particular argument to a particular return type.

3 Algorithm

3.1 Notation

Grammar: These are the types that we work with, and lifetime annotations.

$T ::$ &L `mut` T	A mutable borrow to some type
| &L T	An immutable borrow to some type
| O	A structure or a primitive type
$O ::$ S	A struct or prim type not containing borrows
| S<L>	A struct containing borrows
$L ::$ `'id`	Lifetime names
$S ::$ `id`	

This surface syntax is almost identical to Rust. One important difference is that we assume that all borrow lifetimes are explicitly specified, whereas Rust allows some lifetimes to be inferred or *elided*. Further, we've deliberately considered only structures with one lifetime annotation for simplicity, whereas Rust can have multiple.

Judgements:

$O\{T\}$	"O contains a value of type T as one of its fields"
$T \to T' :\ L$	"T contains another type T' that outlives lifetime L"
$T \to T' :\ \epsilon$	"T contains another *owned* type T'"
$T \to L_1 :\ L_2$	"From T, we can infer that L_1 is longer than L_2"

3.2 Rules for Decomposing a Type

We want to take a type T and extract all of its contained types with their associated lifetimes. We will do this using the following rules.

The simplest case - any owned type contains itself and owns itself.

$$\frac{}{O \to O : \epsilon}\ \texttt{contains-self}$$

If there is a borrow to an owned type, then it has to live at least as long as the lifetime of that borrow.

$$\frac{T \to T_1 : \epsilon}{\&LT \to T_1 : L}\ \texttt{borrow} \qquad \frac{T \to T_1 : \epsilon}{\&L\ \texttt{mut}\ T \to T_1 : L}\ \texttt{mut-borrow}$$

If a structure has a field of type T, then anything that can be extracted from that T can also be extracted from the structure.

$$\frac{O\{T\} \quad T \to T_1 : L}{O \to T_1 : L}\ \texttt{field} \qquad \frac{O\{T\} \quad T \to T_1 : \epsilon}{S \to T_1 : \epsilon}\ \texttt{field-eps}$$

If an extracted type already has an associated lifetime, then any outer borrow lifetime does not attach to it.

$$\frac{T \to T_1 : L_1}{\&LT \to T_1 : L_1}\ \texttt{inner-lifetime}$$

Implicit bounds between two Lifetimes:
From a type given to us, we can infer some things about that type: Any outer borrow lifetime must be shorter than an inner borrow lifetime

$$\frac{T \to T_1 : L_1}{\&LT \to L_1 : L}\ \texttt{longer-lifetime}$$

We can always conclude that a lifetime outlives itself.

$$\frac{}{T \to L :\ L}\ \texttt{outlives-self}$$

Lifetime bounds can also be extracted from inner types.

$$\frac{T \to T_1 : \_ \quad T_1 \to L_1 :\ L_2}{T \to L_1 :\ L_2}\ \texttt{extract-bounds}$$

3.3 Checking Violations

Suppose an argument and return type are of types T_1 and T_2 respectively. Then for all L_1, L_2 such that $T_1 \to T' : L_1$ and $T_2 \to T' : L_2$, $T_1 \to L_1 : L_2$ is not true, this is a possible violation. In notation,

$$\frac{T_1 \to T' : L_1 \quad T_2 \to T' : L_2 \quad T_1 \nrightarrow L_1 : L_2}{\texttt{violation}}\ \texttt{violation}$$

Here we are using the non-standard judgement $T_1 \nrightarrow L_1 : L_2$ to mean that we cannot extract $L_1 : L_2$ from T_1. This is not ideal, so it is just a stop-gap measure.
We report violations to our oracle that checks if there is *actually a transfer* of a value of type T' between T_1 and T_2.

3.4 In the Presence of Raw Pointers

In an ideal world, we want a structure's annotation to be representative of what it holds.

- If a structure is annotated with lifetime `<'a>`, then we would like everything accessible from the structure to outlive `'a`.
- If a structure has no lifetime annotation, then we would like everything accessible from the structure to be owned by the structure object.

But raw pointers break that guarantee. A structure can have lifetime annotation `<'a>`, but the thing pointed to by the raw pointer can live shorter than that. Our plan is to *enforce these guarantees.*

Extending our Rules: If a structure with no lifetime annotations contains a raw pointer, then the pointed-at type acts as though it is owned by the structure object.

$$\frac{S\{\texttt{*const } T\}}{S \to T : \epsilon}\ \texttt{raw-lifetime} \qquad \frac{S\{\texttt{*mut } T\}}{S \to T : \epsilon}\ \texttt{raw-mut-lifetime}$$

If a structure with a lifetime annotation `<'a>` contains a raw pointer, then the data that it points to must outlive `'a`.

$$\frac{S\{\texttt{*const } T\}}{S\texttt{<}L\texttt{>} \to T : L}\ \texttt{raw-owned} \qquad \frac{S\{\texttt{*mut } T\}}{S \to T : L}\ \texttt{raw-mut-owned}$$

Checking for violations is the same process, but with the extended set of inference rules.

4 Implementation

I have implemented the algorithm described in this report using the Rust High-level IR and Mid-level IR (code in footnote)[4]. This restricted set of types and syntax presented in this report are unfortunately not general enough to capture the complexity of real-world Rust code, so the tool is a lot more complicated than just this approach described here. Nevertheless, the basic structure and principles are the same.

5 What is yet to be done

I hoped to show some examples of function signatures/types that fit the rules we described here as well as some that didn't, along with the derivations. I also hoped to show some preliminary results of running my tool on Rust code to detect violations.

[4] `https://github.com/vikramnitin9/Rudra`

However I do not have time to include all this in the report, so I am making this submission before the deadline with the material that I currently have.

Post-deadline, I shall augment this report with more results and discussion.