Project Proposal for TLC

Name: Vikram Nitin
UNI: vn2288
Group: Would like to work solo (see last paragraph of doc)

As part of my research, I've been working on a project to build a static analysis tool to detect certain types of memory safety bugs in unsafe Rust. Rust offers strong memory safety guarantees thanks to its borrow checker, and a key aspect of borrow checking is determining whether a borrow can outlive the memory it points to. If so, that is dangerous and should not compile. To track this, the borrow checker associates a "lifetime", or a concrete region of code, to each instance of a type, and propagates these lifetimes across code.

However, Rust makes a deliberate design choice not to infer certain lifetimes of parameters in function signatures. In such cases, Rust requires manual annotation of types with "lifetime parameters". This looks something like this:

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str
```

where the `'a` is a lifetime annotation. This means that the compiler has to find some concrete region of code `'a`, such that `x`, `y` and the returned borrow (`&'a str`) do not outlive `'a`. This region `'a` becomes the lifetime of each of the borrows.

Now for safe Rust code, the compiler can still statically infer all lifetimes, and it will warn you if the lifetimes it infers are not compatible with the lifetime annotations. However for unsafe Rust, the lifetime annotations can make or break your code, because there are some lifetimes that the compiler can no longer infer. Incorrect lifetime annotations on function signatures can lead to use-after-free or aliased mutability bugs.

I've developed a static analysis tool to detect incorrect lifetime annotations on function signatures. The key idea is to check for dataflow between short-lived data and a longer-lived reference, by extracting lifetime annotations associated with each type in the signature.

For example, if we have a function

```
fn get<'a, 'b>(&'a self) -> &'b T {
    unsafe{&*self.data}
}
```

then this is suspicious because `self` is borrowed for a lifetime `'a`, but we're returning a reference to `self.data` with an unrelated lifetime `'b`. It's possible that `self.data` might be freed (or "dropped") while the returned reference is still valid.

The goal of my project for this course would be to codify these rules in terms of the formalisms we have been studying. While tracking lifetimes through function bodies is hopelessly complex, I think a simple manageable goal would be:

Say that a pair of types A and B are "safe" if A must outlive B. Then it is safe for B to hold the data from A. Here, a "type" consists of both the datatype (i32, String, Vec, etc) as well as the lifetime annotation.

Say that a function signature is "well-typed", if every pair of types in the function signature is safe. This is in effect saying that we don't care about the body of the function - we want a worst-case guarantee.

The goal would be to define what "safe" and "well-typed" mean in formal terms. I would have to restrict my types to some subset of the types in Rust to make it manageable.

For the programming component of this project, I will complete my half-finished tool to detect lifetime violations. The tool will be an extension of the theory that I develop above.

I would like to work on this project alone, because I feel it would be hard to collaborate effectively with another person on this. Some reasons are:

1. I've been studying lifetimes in Rust for a while now, and I feel like it would be hard to find someone else with the same context.
2. I already have a half-written tool to detect lifetime violations which I'd like to complete and include as part of the project

Some prior work in the area:

1. Oxide [1] models Rust code with non-lexical lifetimes and a syntax close to safe surface Rust. However it does not model unsafe code.
2. RustBelt [2] develops a comprehensive formal semantics based on lifetimes for a realistic subset of Rust including unsafe code.
3. FR [3] is a lightweight formalism of Rust that captures type checking and borrow checking.

[1] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2019. Oxide: The essence of rust. arXiv preprint arXiv:1903.00982 (2019).
[2] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. Proceedings of the ACM on Programming Languages 2, POPL (2017), 1–34.
[3] David J Pearce. 2021. A lightweight formalism for reference lifetimes and borrowing in Rust. ACM Transactions on Programming Languages and Systems (TOPLAS) 43, 1 (2021), 1–73.