AUTOMATICALLY DETECTING LIFETIME ANNOTATION BUGS IN THE RUST LANGUAGE

Vikram Nitin*, Sanjay Arora†, Anne Mulhern†, Baishakhi Ray*

* - Columbia University, † - Red Hat Research
vikram.nitin@columbia.edu, {saarora, amulhern}@redhat.com, rayb@cs.columbia.edu

WHY RUST?
Efficient performance while providing strong memory safety, thread safety, and type checking.

SOME FEATURES OF RUST

Ownership
Move
Transfer Ownership
a is no longer valid; accessing it gives error

Borrowing
b is borrowing w/o transferring ownership
a is valid; accessing it does not give error

BORROW CHECKING AND LIFETIMES

RUST uses lifetimes, a construct that assists the compiler in verifying the validity of each borrow. Both borrow variables and the borrowed values have lifetimes, as illustrated below.

RAW POINTERS AND UNSAFE CODE

RUST’s borrow checker can be too restrictive sometimes because its analysis is inherently conservative. So RUST provides “raw pointers”(*const, *mut) that don’t have borrow checking.

Dereferencing raw pointers requires an unsafe block

LIFETIME ANNOTATIONS

Consider a function that takes two borrows as inputs and returns one borrow.

```rust
fn foo(x: &i32, y: &i32) -> &i32 { /*...*/ }
```

The two input borrows could have different lifetimes, in which case an appropriate lifetime must be assigned to the output borrow. It could be annotated as follows, indicating that the returned borrow refers to x.

```rust
fn foo<'a,'b>(x: &'a i32, y: &'b i32) -> &'a i32 { /*...*/ }
```

Here 'a and 'b are lifetime annotation parameters. At compile-time, the compiler assigns concrete scopes to each lifetime parameter. Appropriate lifetime annotations can guide the borrow checker to ensure memory safety. Consider the following example:

```rust
fn foo() {
    let a = String::from("Hello");
    let b = &a;
    println!("{}"), a);
}
```

The lifetime of a borrow variable cannot be longer than the lifetime of the borrowed value. The following code will not compile because the borrow checker will flag an error.

```rust
fn foo() {
    let a = String::from("Hello");
    let b = &a;
    println!("{}"), a);
}
```

The implicit borrow to `foo` on line 6 needs to last till line 8, but the `foo` object is valid only until line 6. So this will raise a compilation error, which is good.

OUR SYSTEM

When RUST code is compiled, it goes through a High-level Intermediate Representation (HIR) and a Mid-level Intermediate Representation (MIR). We implement a hybrid analysis that combines information from the HIR and the MIR, as shown in the diagram alongside.

We implement our system as a subroutine within the Rudra project.

Our code will soon be made publicly available.

BUG PATTERNS

Incorrect lifetime annotations on functions that handle raw pointers can cause memory safety violations. We consider the specific case of structures containing raw pointers.

Consider a function that manipulates a raw pointer inside a structure. Then there are three broad categories of manipulations that involve a borrowed value. The function could:

1. Take a borrow as input and update the raw pointer to point to the borrowed value.
2. Read from the raw pointer and return a borrow to the value that it’s pointing to.
3. Take a borrow as input and create a new structure object with the raw pointer pointing to the borrowed value.

We define three patterns of bugs based on these three operations. A full discussion of the three patterns is beyond the scope of this poster, but we discuss the first pattern here.

The function `foo` takes an input borrow `inp` with lifetime `'a`, but the lifetime of the structure object is `c`. Further, there is a `dataflow` from `inp` to `self.inner`. This means that the structure could potentially outlive the borrowed value.

```
let i = "Hello".to_string();
let mut obj = Foo{inner: i};
```

BORROW CHECKING AND LIFETIMES

RUST uses lifetimes, a construct that assists the compiler in verifying the validity of each borrow. Both borrow variables and the borrowed values have lifetimes, as illustrated below.

LIFETIME ANNOTATIONS

Consider a function that takes two borrows as inputs and returns one borrow.

```
fn foo(x: &i32, y: &i32) -> &i32 { /*...*/ }
```

The two input borrows could have different lifetimes, in which case an appropriate lifetime must be assigned to the output borrow. It could be annotated as follows, indicating that the returned borrow refers to x.

```
fn foo<'a,'b>(x: &'a i32, y: &'b i32) -> &'a i32 { /*...*/ }
```

Here 'a and 'b are lifetime annotation parameters. At compile-time, the compiler assigns concrete scopes to each lifetime parameter. Appropriate lifetime annotations can guide the borrow checker to ensure memory safety. Consider the following example:

```
fn foo() {
    let a = String::from("Hello");
    let b = &a;
    println!("{}"), a);
}
```

The lifetime of a borrow variable cannot be longer than the lifetime of the borrowed value. The following code will not compile because the borrow checker will flag an error.

```
fn foo() {
    let a = String::from("Hello");
    let b = &a;
    println!("{}"), a);
}
```

The implicit borrow to `foo` on line 6 needs to last till line 8, but the `foo` object is valid only until line 6. So this will raise a compilation error, which is good.

OUR SYSTEM

When RUST code is compiled, it goes through a High-level Intermediate Representation (HIR) and a Mid-level Intermediate Representation (MIR). We implement a hybrid analysis that combines information from the HIR and the MIR, as shown in the diagram alongside.

We implement our system as a subroutine within the Rudra project.

Our code will soon be made publicly available.

BUG PATTERNS

Incorrect lifetime annotations on functions that handle raw pointers can cause memory safety violations. We consider the specific case of structures containing raw pointers.

Consider a function that manipulates a raw pointer inside a structure. Then there are three broad categories of manipulations that involve a borrowed value. The function could:

1. Take a borrow as input and update the raw pointer to point to the borrowed value.
2. Read from the raw pointer and return a borrow to the value that it’s pointing to.
3. Take a borrow as input and create a new structure object with the raw pointer pointing to the borrowed value.

We define three patterns of bugs based on these three operations. A full discussion of the three patterns is beyond the scope of this poster, but we discuss the first pattern here.

The function `foo` takes an input borrow `inp` with lifetime `'a`, but the lifetime of the structure object is `c`. Further, there is a `dataflow` from `inp` to `self.inner`. This means that the structure could potentially outlive the borrowed value.

```
let i = "Hello".to_string();
let mut obj = Foo{inner: i};
```

Acknowledgements: This research was supported by a grant from Red Hat Research