CIS 561 - Paper Review

YUYA KAWAKAMI

CIS 561: Introduction to Compilers

1 INTRODUCTION

We review Pearce's *A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust* [1]. The intention for this review is not to review the formalization that Peace presents and its correctness. Rather, the goal for this review to understand the following question - "How does Rust enable compile-time garbage collection?" The short answer to the above is the following. Rust enables compile-time garbage collection through a strict ownership model. At a basic level, any object on the heap has one and only one owner. Other things can make references to it, but it has one owner. Under this setup, Rust can automatically free memory, by recursively dropping memory when the owners go out of scope.

Of course, the additional detail is that Rust also needs to be able to figure out the lifetimes of variables. Although in the basic setup that Pearce presents, this corresponds well to the curly braces for primitives.

We will first present some examples that Pearce uses to highlight some of Rust's capabilities

2 RUST

A key component that allows Rust's compile-time garbage collection is, as mentioned prior, the ownership design. In essence, Rust enforces that no two variables can own the same value. Pearce introduces this as *Ownership invariance*. Though, this is slightly relaxed when in the context of borrowing, the ownership design allows Rust to safely drop and deallocate any location that a variable *owns*, when that variables goes out of scope. Note that allowing exactly one owner allows this - we can be sure that this doesn't 'break' some other variable.

Another key feature is Rust's borrowing concept. Borrowing allows members to access the shared resources, but in language like C, one often counters race conditions. Especially for HPC applications, where many threads often access and write into shared resources, avoiding race conditions is critical. However, in Rust, the language is designed such that no two *mutable* borrowed reference can exist in one lifetime. Distinguished by `&mut x, &x`, Rust draws a line between references through which the original variable can be altered and those that cannot. By default, references are immutable - speaking to the safety-centered design of Rust.

Determining the lifetimes of variables is a powerful tool that Rust provides. Consider the following snippet:

```
fn f(p : &i32) -> &i32 {
    let x = 1;
    let y = &x;
    return y;
}
```

As Pearce describes, this is a classic example of returning a reference to a stack-allocated data. C for example, will allow the equivalent code with no complaint. However, Rust understands that the lifetime of `x` is longer than that of `y`; hence, rejecting the program.

REFERENCES

[1] David J. Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Trans. Program. Lang. Syst.* 43, 1, Article 3 (apr 2021), 73 pages. https://doi.org/10.1145/3443420