PLPDI - Assignment 4

Sam Barrett, 1803086

December 15, 2020

For my report, I chose to focus on the paper *Safe Systems Programming in Rust: The Promise and the Challenge*[1]. In this paper, Jung et al. discuss the promises made by the Rust programming language in terms of memory safety through the use of types. They go on to sketch a system by which these claims can be verified through the use of formal logics and proof systems. They title their ongoing project *'RustBelt'*[2].

The largest challenge they envisage is reasoning about the interaction between the safe and unsafe fragments of the Rust language. Common techniques for ensuring memory safety of programs rely on a method known as *Syntactic type soundness*. However, due to the unsafe nature of many underlying Rust APIs this system is simply demonstrated to not be powerful enough. Given a known safe program `e` if we wrap this with semantically sound code to produce: `if true { e } else { crash() }` we now have syntactically unsafe code. Because, in Rust, this is perfectly memory safe, as the `crash()` is unreachable, we therefore cannot use *Syntactic type soundness* to prove the memory safety of Rust. Jung et al. then turn to *Semantic type soundness*, employing *Iris* logic implemented using the Coq theorem prover.

Being able to formally prove the safety of one's code is extremely important for high security systems or systems of high import. For example, in a system that monitors aeroplane sensors any crash, bug or insecurity could be catastrophic, possibly resulting in the loss of lives or expensive equipment. Being able to prove that such a system is immune from crashes, bugs or exploitation is a very attractive property of a language.

This paper primarily employs the concepts discussed in the *Principals of Programming Languages* section of the module, specifically the types subsection. However, it also touches on the model checking subsection of the *Formal Verification* section.

The underlying system (Iris) for proving *Semantic type soundness* will employ typing rules as seen in Lecture-7 to infer the types of functions and variables. *Iris* employs an ML-style language along with a form of Hoare logic to encode the rust APIs into their respective safety contracts. Once in this form the overlying Coq theorem assistant can determine whether the code adheres to the contract or not, i.e. whether the code is *safe* or *unsafe*.

This method of representation is shared with that we used when studying the Simply Typed λ-Calculus as both Iris and the λ-Calculus share a foundation in mathematical logic.

One weakness of the approach taken is that it appears that the method of proving soundness of an internally unsafe Rust library is not automated, requiring someone with a extensive background in logic and theorem proving. The author made clear that this is an area of current development but it still renders the current system insufficient for proving the memory safety of all rust programs.

The paper mentions the facility of *Iris* to support *'impredicative invariants'* which allow for cyclic dependencies or invariants that reference other invariants. They specifically mention this being useful for modelling recursive types and closures within Rust. However, to my knowledge, Rust does not support recursive types specifically due to the concerns of memory safety, the compiler does not know how much memory is required at compile time; this can be subverted via the use of smart pointers through the `Box<T>` structure and perhaps *impredicative invariants* might be used to model this approach. Rust's inability to implicitly allow recursive types is demonstrated in Figure 1.

The work outlined in this paper is a part of a ongoing project and as such this paper does not display any concrete results, therefore, the success cannot be evaluated. The potential of the project is great; with many industries looking for reliability and security along with performance when choosing a language, if this project succeeds Rust will be an extremely attractive option with C-like performance, secure and robust code and modern features such as closures and pattern matching.

```
        enum Foo {
//      ^^^^^^^^ recursive type has infinite size
             Bar(Foo)
//                --- recursive without indirection
        }

        fn main() {
             println!("You cannot have recursive types in Rust");
        }
// help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make `Foo` representabl
//   |
//3  |      Bar(Box<Foo>)
//   |          ^^^^    ^
```

Figure 1: Recursive types in Rust

References

[1] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Safe Systems Programming in Rust: The Promise and the Challenge. *Communications of the ACM*, 2020.

[2] RustBelt. https://plv.mpi-sws.org/rustbelt/.