Computing Fine-grained Type Information for Rust Programs

Project Description

William Seddon

Supervisors: Federico Poli, Vytautas Astrauskas, Prof. Peter Müller

30 March 2020

1 Introduction

Rust is a rapidly maturing programming language which improves upon systems languages such as C and C++ by introducing new safety guarantees and features. It uses an innovative ownership type system which provides extensive safety guarantees, protecting against common issues such as data races and dangling pointers. To achieve these safety guarantees the type system imposes extra rules which languages such as C++ do not have. For a tool to leverage Rust's powerful type system when reasoning about Rust programs, it requires more information than the Rust compiler currently exposes. The scope of this project is to develop techniques and implement a library to provide this additional fine-grained type information to third-party tools.

2 Problem

In an ownership type system every value has an owner [2], which in Rust is a variable. The ownership of a value can be temporarily transferred by borrowing, permanently transferred by moving, or the value can be duplicated by cloning it. To provide Rust's safety guarantees the type system imposes rules, for instance only one mutable borrow, also known as a mutable reference, to a certain value can exist at any one time. These rules are checked by the compiler's type checker and borrow checker. These restrictions cause syntactically valid, visually correctly typed, in-scope use of variables to be invalid, an example of this is shown in figure 1. In Rust the type used in the definition of a variable is no longer enough to be sure that a variable access is allowed. Rust tooling requires more fine-grained type information which keeps track throughout the program of which variables are allowed to be accessed in accordance with the rules of the type system.

One such tool is Prusti, a tool developed at the ETH Zürich to formally verify Rust programs using the Viper verification infrastructure. To address the

```rust
let a = String::from("Friends, Romans, countrymen, lend me
    ↪ your ears");
```

```rust
// more code ..

// a is of type String
println!("Anthony: {}", a);

// OK
```

```rust
let b = a; // a moved out

// more code ..

// a is of type String
println!("Anthony: {}", a);

// Error!
```

Figure 1: Rust code comparison showing how context affects if variables can be accessed

problem of lacking information the Prusti paper [1] proposes a novel fine-grained type information concept: place capabilities and place capability sets which track exactly which variables can be accessed throughout a Rust program. A place capability is the ability to access a certain variable, and the place capability set is the set of place capabilities at a certain point in the program. An example program annotated with the appropriate place capability sets can be found in figure 2. While the Prusti paper introduces the notion of place capability sets, their exact semantics and how they are generated is left undefined. The scope of this thesis is to provide a specification for place capability sets and an implementation of a useful library for Rust tools. The use of place capability sets is necessary for Prusti to be sure if an access to a variable at a certain point is allowed.

3 Project Approach

Any tools reasoning about Rust programs, not just Prusti, need a library which calculates and exposes the fine-grained type information. In this case the library will build place capability sets and allow tools to access the information. The library needs to extract information from the Rust compiler, utilising both type checker and borrow checker state [4] to calculate and build place capability sets. It should then expose the place capability sets in a standard format to a variety of tools.

Wolff [5] worked on an extension of place capability sets called extended place capability sets (EPCS) in the report *Extended Place Capabilities Summaries for Rust Programs*. The extension to place capability summaries lay in devising a method to keep track of references. The approach outlined in the report gives a definition of EPCS and includes the small-step semantics for the generation of these extended place capability sets from the Rust control flow graph. There are however areas that are left unexplored, for example when dealing with loops an oracle is required to give the loop invariant. The EPCS implementation also only supports a small subset of the Rust language, handling assignments, which makes it difficult to use on real-life code. Areas such as the description of the

```rust
let a = String::from("Cowards die many times before their
    ↪ deaths"); // {a}

// Move
let mut b = a; // {b}

// Mutable Borrow
{
    let c = &mut b; // {c}
    c.push_str("; The valiant never taste of death but once");
} // {b}

// Immutable Borrow
let d = &b; // {b, d}

// Clone
let e = d.clone(); // {b, d, e}

println!("Caesar: {}", e); // {b, d, e}
// Caesar: Cowards die many times before their deaths; The
    ↪ valiant never taste of death but once
```

Figure 2: Example Rust code showing the place capability sets as comments

interaction with and the requirements of the borrow checker and Polonius [3] could be improved.

This is a big field with a lot of different cases; this project should improve upon some common cases not explored by this previous work.

4 Core Goals

The aim of this project is to create a prototype reusable library which provides this needed fine-grained type information to tools and applications. How the place capability sets are generated and defined must rest upon solid theoretical foundations and definitions. From this come the four core goals:

1. To understand the current information given by the various parts of the Rust compiler such as the type-checker and borrow-checker, while documenting what information is exposed as well as documenting any improvements that could be made at the Rust compiler level.

2. To design an algorithm which takes the information gleaned from the Rust compiler and processes it into a place capability set. This involves defining the semantics of a place capability set that is designed for common Rust code cases, as well as various algorithmic steps to convert the compiler information into the place capability set format. The Rust cases that the algorithm should cover includes comparatively complex language features not explored by the previous work such as method calls with lifetime arguments and loops of varying degrees of complexity.

3. To implement a library which implements these algorithms and exposes the more fine-grained type information of Rust programs to users of the library. Where appropriate the library should be based upon the previous extended place capability implementation. The library should work with a wide range of Rust code, meaning the library should support language features the previous implementation does not. These features, for example, include branches, the creation of types, method calls and common loop cases.

4. To evaluate the implementation of the library by creating a simple command-line front-end which clearly displays the extra type information gleaned from the algorithm.

5 Extension Goals

The aim of the extension goals is to provide examples of how useful this extra fine-grained type information is by developing tools that depend on this new information. As such the extension goals are:

- To develop a code completion tools for IDEs to have the code completion suggest variables that are valid when used at that current location.
- To implement an IDE plug-in which integrates the fine-grained type information into the programmer's view of the IDE.
- To integrate the library into Prusti. Currently Prusti uses a sub-optimal implementation of place capability sets which would be improved by integrating this new library.
- To create a type-checker-checker tool which creates a checkable step-by-step representation of what the type-checker checks, as well as a small prover tool which checks this proof tree.
- To extend or implement a fuzzer which utilises the knowledge of which variable fields can be accessed at any point to generate correct Rust programs which can be used to test tools such as Prusti.
- To create a tool which analyses Rust code to calculate which types definitions are needed to reason about a function.

References

[1] V. Astrauskas et al. "Leveraging Rust Types for Modular Specification and Verification". In: *ACM Program. Lang., Vol. 3, No. OOPSLA, Article 147* (2019). `http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=AstrauskasMuellerPoliSummers19b.pdf`.

[2] The Rust Book Contributors. *Rust Book Chapter 4*. URL: `https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html`.

[3] N. Matsakis. *An alias-based formulation of the borrow checker*. 2018. URL: `http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/`.

[4] N. Matsakis. *Introducing MIR*. 2016. URL: https://blog.rust-lang.org/2016/04/19/MIR.html.

[5] D. Wolff. *Extended Place Capabilities Summaries for Rust Programs*. 2019. URL: https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Dylan_Wolff_RICS_Report.pdf.