Ownership Typesystem based Optimisations for Rust

Master Thesis Project Description

Janis Peyer
Supervised by Prof. Dr. Peter Müller, Vytautas Astrauskas, Federico Poli
Department of Computer Science
ETH Zürich
Zürich, Switzerland

I. Introduction

Rust [1] is a modern programming language designed to develop performant and safe low-level software. Recent work [2] defines an operational semantics for Rust called Stacked Borrows, which justifies new optimisations by introducing undefined behaviour. So far these optimisations were not implemented in Rust. Implementing static analyses that enables these optimisations is the main goal of this master's thesis.

In this section we will first describe the kind of optimisations we want to implement and briefly explain the Rust knowledge required to understand them, and last we will talk about Stacked Borrows and explain how it justifies these optimisations.

A. Optimisations

The optimisations we want to implement are based on the knowledge, that certain references do not alias each other. Two references alias each other if the location they point to partially or fully overlap. To explain the target optimisation and how non-aliasing information makes it possible we will discuss a concrete example right away.

```
1  fn no_alias(
2      x: &mut i32,
3      y: &mut i32,
4  ) -> i32 {
5      *x = 42;
6      *y = 7;
7      return *x;
8  }
```

Listing 1: Example of a Non-Aliasing Optimisation

Listing 1 defines a function `no_alias` that takes two mutable references `x` and `y` as parameters. Then in the body of the function we first write `42` to the underlying value of `x`, second we write `7` to the underlying value of `y`, and last we return the underlying value of `x` as the result of the function.

We would now like to argue, that `x` and `y` do not alias each other. If that was the case, we could show that the assignment to `*y` does not change the underlying value of `x` and therefore, we could optimise the code above to return `42` directly instead of reading `*x` again. Because `x` and `y` are mutable references, Rust prevents them from aliasing each other. This is one of the rules that are enforced at compile time by a static stage of the Rust compiler called borrow checker.

While the borrow checker has a lot of advantages and ensures memory safety, there are cases where the enforced rules are too restrictive and prevent valid use cases. An example of a valid use case that is not allowed, is a channel data structure that allows inter-thread communication, because it requires shared mutable state. Another even simpler example is building a cyclic linked list, which would also be prevented by the borrow checker.

This is where unsafe Rust comes into play. Unsafe Rust is a programming language similar to (safe) Rust, but it allows accessing shared mutable state and the use of C-style raw-pointers including pointer arithmetic. Unsafe Rust can be used inside of a safe Rust code-base by specifying scopes as unsafe. Code inside of such a scope is handled as unsafe Rust.

Users are advised to predominantly use safe Rust and either use unsafe Rust in concise small blocks of code or by using modules that provide safe abstractions around unsafe code. Even with these guidelines, unsafe Rust has been found to be used in a significant amount of Rust projects [3], so we have to consider unsafe Rust when we develop optimisations for Rust.

Using unsafe Rust we can construct an example that would make the shown optimisation of replacing `*x` with `42` in Listing 1 invalid. The following Listing 2 shows, how unsafe Rust can be used to create two mutable references that point to the same underlying value and pass them to our function `no_alias` that is completely written in safe Rust.

On line 2 a pointer to the local variable `v` is created. Then `no_alias` is called, and using the pointer two mutable references that both point to `v` are passed as arguments. Because pointer dereferencing is used, the function call has to be wrapped in an `unsafe` block to opt-in to unsafe Rust. This example compiles without errors and is considered valid by the borrow checker.

This program would be a counterexample to the outlined optimisation: for our function `no_alias`, the mutable

```
let v = 5;
let raw_pointer = &mut v as *mut i32;
let result = unsafe {
    no_alias(
        &mut *raw_pointer,
        &mut *raw_pointer,
    )
};
```

Listing 2: Unsafe Rust Code Creating Aliasing Mutable References

references `x` and `y` could potentially alias each other. That would mean the assignment to `*y` could also modify the underlying value of `x` and we could not simplify the subsequent read from `*x`.

B. Stacked Borrows

Stacked Borrows [2] is an operational semantics which encodes aliasing rules for references in safe and unsafe Rust. These rules are designed to compute identical or more liberal rules than Rust's borrow checker for safe Rust, but other than the borrow checker, Stacked Borrows' rules also extend to unsafe Rust.

The Stacked Borrows rules are implemented in Miri, which is an interpreter for Rust that works as a dynamic analysis tool for aliasing information when combined with Stacked Borrows. Because Miri is an interpreter, executing a program is about 1000 times slower compared to running the same code as a compiled binary. That is the main reason, why the interpreter is implemented to run nightly test suites rather than replacing compiled rust.

Stacked Borrows work by tracking which references currently have access to which memory locations. Non-reference variables are treated as references to their stack locations. Each memory location tracks which references currently have access on a borrow stack: Creating a reference pushes it on the stack. Accessing the value with reference r pops every reference above r from the stack. This is introduced as stack principle, which enforces well-nested usage of references and is argued to be equivalent to what the static borrow checker does but extended to unsafe Rust and C-style pointers.

For the use in optimisations, everything that violates Stacked Borrows' rules is considered undefined behaviour. Doing this justifies, that the compiler can assume an input program to conform to Stacked Borrows in every situation and is free to do anything if it does not. Using this back in our optimisation example, we can now formally reason that Listing 2 does not conform to Stacked Borrows and introduces undefined behaviour. Therefore, our optimisation of replacing the read from `*x` with the constant `42` in Listing 1 would still be valid, even when used like shown in Listing 2.

II. Main Goal

The main goal for this master's thesis is to design a static analysis usable for optimisations. The analysis will assume that the input code passes the borrow checker and conforms to Stacked Borrows. The output of this analysis is then used by optimisations that we develop to improve the runtime of the input code. Both analysis and optimisations will work intra-procedurally and will be created as extension to the Rust compiler.

The following sections describe optimisations that will be implemented during the thesis. These optimisation ideas originate from the Stacked Borrows paper [2] and we introduce the different optimisations here in the same order as they appear in the paper. The plan is to also implement the optimisations in this order.

A. Move Up for Mutable References

The first optimisation will only consider mutable references. More concretely, we want to *move up* reads from mutable references across instructions that will never mutate the value. The following example in Listing 3 illustrates this. We previously used the same example in the introduction.

```
1 fn mutable(
2     x: &mut i32,
3     y: &mut i32,
4 ) -> i32 {
5     *x = 42;
6     *y = 7;
7     return *x;
8 }
```

Listing 3: `mutable` Original Input Code

`*x` is written to on line 5 and read again on line 7. `x` and `y` could potentially point to the same value, but a trace, where that is the case, would not conform to Stacked Borrows and therefore would be undefined behaviour. So, under our assumptions `x` and `y` do not alias and the read can be moved up past the write to `*y` on line 6. The following Listing 4 shows the code after performing this transformation:

```
1 fn mutable(
2     x: &mut i32,
3     y: &mut i32,
4 ) -> i32 {
5     *x = 42;
6     let val = *x;
7     *y = 7;
8     return val;
9 }
```

Listing 4: `mutable` Transformed Code

So far we only moved the read, but since the write and read are now directly consecutive, the second read can be

replaced by the constant used in the write. Since we used a fresh variable (not used anywhere else) to hold the return value, the variable can be eliminated and the value can be directly returned. This finally results in the following optimised code:

```
fn mutable_opt(
    x: &mut i32,
    y: &mut i32,
) -> i32 {
    *x = 42;
    *y = 7;
    return 42;
}
```

Listing 5: `mutable_opt` Optimised Code

B. Move Up for Shared References

The next optimisation will extend the support for the *move up* to also encompass shared references. Shared references only allow read-only access and if the program conforms to Stacked Borrows, the underlying value cannot be changed between two reads from a shared reference.

The following Listing 6 shows an example of where a *move up* optimisation would be useful when using a shared reference. The reference `x` is twice read from and divided by 3. Note that even if we pass the reference `x` into another function, the underlying value is still guaranteed to be unmodified after that function returns.

```
fn shared(x: &i32) -> i32 {
    let val = *x / 3;
    f(x, val);
    return *x / 3;
}
```

Listing 6: `shared` Original Input Code

The next Listing 7 shows the result after the optimisation. We moved the expression `*x / 3` up across the function call and then simplified the code to only execute the expression once, getting rid of a needless read and division.

```
fn shared_opt(x: &i32) -> i32 {
    let val = *x / 3;
    f(x, val);
    return val;
}
```

Listing 7: `shared_opt` Optimised Code

C. Move Down using Protectors

To support optimisations which *move down* reads we have to rely on Stacked Borrows' protectors in the general case. The following Listing 8 shows the function `protectors` in which we would like to move the read access of `x` down across the function calls to reduce register pressure (amount of registers required at once).

```
fn protectors(x: &i32) -> i32 {
    let val = *x / 3;
    f1(x);
    f2();
    return val;
}
```

Listing 8: `protectors` Original Input Code

It is important to note, that in this example protectors are required. Protectors are a technique in Stacked Borrows to make sure that function parameters outlive the function call. In the example this is relevant because the lifetime of `x` could otherwise potentially end after the call to `f1` and the underlying value of `x` be modified through a pointer in `f2` while still conforming to Stacked Borrows; protectors prevent that and would render such a trace invalid.

The following Listing 9 shows the function after the optimisation of moving access to `x` down.

```
fn protectors_opt(x: &i32) -> i32 {
    f1(x);
    f2();
    return *x / 3;
}
```

Listing 9: `protectors_opt` Optimised Code

D. Evaluation

This subsection briefly details how the implementation is going to be evaluated.

- **Unit Tests:** Evaluate that the optimisations are applied as expected to a set of crafted examples. The examples and test suite should ideally be created before the implementation.
- **Real World Code:** Apply the developed optimisations to existing code bases. Evaluate, where the optimisations got applied and run the unit tests for those projects after applying optimisations. If the optimisations get applied a lot and no unit tests break, this gives some measure of confidence that the applied transformation do not introduce breaking changes. If none or only a few optimisations get applied this could indicate, that we should broaden the optimisation or test on different code bases.
- *(Optional)* **Benchmarking:** Benchmark the compilation time with and without the optimisations. One hypothesis is, that compilation time might be shorter when adding our optimisations, because the LLVM (Rust back-end compilation) optimisations get simplified. We could also create a benchmark to compare the runtimes of code with our optimisations applied to the unoptimised version of the same code.

III. Core Goals

The main goal above is organised into the following core goals.

- **Craft Examples:** Create code examples for the different optimisations and possibly combine them into an automatic test suite. The examples should include code that is expected to be optimised, as well as code that is not allowed to be optimised for every implemented optimisation.
- **Gather Optimisation Requirements:** Discover what information is required to implement the desired optimisations.
- **Design Static Analysis:** Design the details of the non-aliasing static analysis.
- **Expose Static Analysis:** Expose the static analysis in a way consumable by the optimisations.
- **Implement Optimisations:** Implement the two "move up" optimisations II-A and II-B.
- **Evaluation:** Evaluate the developed optimisations (II-D).

IV. Extension Goals

This section explains extension goals that are possible additions to the thesis after the core goals are completed.

- **Develop Additional Optimisations:** Design, implement and evaluate additional optimisations that use aliasing-information. Extend the static analysis if needed. This could involve but is not restricted to the following points:
  - Implement the "move down" optimisation that depends on protectors described in the main goal section.
  - Automatically replacing vectors of structs with multiple vectors. This would be an optimisation, because operating on multiple vectors of primitive types generally performs better than on a single vector containing a complex type.
    For example, the type A in the following code would be transformed into the type B:
    ```
    type A = Vec<S>;
    struct S { f1: i32, f2: u64 }
    struct B { b1: Vec<i32>, b2: Vec<u64> }
    ```
  - Vectorisation: Knowing Non-aliasing information has the potential to be useful in vectorisation optimisations, where transformations often involve computing multiple results at once instead of computing them one-by-one. This could also be combined with the transformation above, as that also favours vectorisation.
- **Correctness Proof:** Formally prove, that the performed optimisations do not change program behaviour under the assumption, that the program conforms to the Stacked Borrow rules.
- **Alias Information for Function Returns and Nested References:** In the core goals only references passed as parameter were considered. If an aliasing with any other kind of reference was possible, optimisation was not performed. With this extension goal, we also want to compute alias information for references returned by function calls and references obtained by using a dereference.

References

[1] S. Klabnik and C. Nichols, *The Rust Programming Language (Covers Rust 2018)*. San Francisco, CA: No Starch Press, Aug. 2019. [Online]. Available: https://doc.rust-lang.org/book/

[2] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, "Stacked borrows: An aliasing model for rust," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 1–32, Jan. 2020.

[3] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?" *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–27, Nov. 2020.