Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Ownership Typesystem based Optimisations for Rust

Master Thesis

Janis Peyer

July 12, 2023

Advisors: Prof. Dr. Peter Müller, Federico Poli, Vytautas Astrauskas

Department of Computer Science, ETH Zürich

Abstract

Rust is a programming language that provides a high degree of memory-safety. A substantial part of Rust's memory-safety originates from its ownership model which is enforced by the borrow checker. Rust is partitioned into a safe and an unsafe subset and Rust's memory-safety guarantees are only provided for the safe subset of Rust.

Recent work introduces an operational semantics for memory accesses in Rust called Stacked Borrows. This operational semantics encodes the borrow checker rules and makes them applicable to unsafe Rust too. Stacked Borrows enables new optimisations, however, there is currently no work leveraging Stacked Borrows to implement these kinds of optimisations.

In this work we attempt to fill this gap by implementing a static analysis that approximates Stacked Borrows. This static analysis generates information which we call immutability spans. We demonstrate the relevance of these immutability spans by creating an optimisation based on them which can apply improving changes to Rust programs that were previously not performed by the Rust compiler infrastructure.

We evaluate our static analysis and optimisation using our own test suite of manually written programs. Moreover, we also perform an evaluation by running the analysis and optimisation on the official rustc test suite.

Acknowledgements

I would like to express my sincere gratitude to my supervisors Federico Poli and Vytautas Astrauskas. They were always supportive, guided me to overcome challenges, and encouraged me to do my best in this thesis.

I would also like to thank the Programming Methodology Group and in particular Prof. Dr. Peter Müller for allowing me to write my thesis in this group. The feedback and discussion during my intermediate representations were genuinely helpful to not only improve my work but also learn how to explain it in an easily understandable way. Moreover, I really enjoyed the lectures by Prof. Müller and count them towards my absolute favourites during my time at ETH Zürich.

Lastly, I would like to thank my father, Martin Peyer, for his support and our weekly lunches together. They were truly helpful to stay focused on making—at least some small—progress every week and additionally a welcome and fun break from my daily routine.

Contents

Chapter 1

Introduction

Rust [13] is a modern programming language designed to develop performant and safe low-level software. Recent work [12] defines an operational semantics for memory accesses in Rust called Stacked Borrows. In that work Stacked Borrows is used to justify new optimisations that are currently not performed by the Rust compiler. Implementing static analyses that enable these kinds of optimisations is the main goal of this master's thesis. Moreover, our goal is to also implement optimisations that make use of our static analysis to show their relevance.

In this chapter we will briefly describe the kind of optimisations we want to implement and the static information they require, then present the goals of the thesis and give an outline to this report.

1.1 Optimisations

The optimisations we want to enable are based on alias information. Two references alias each other, if the location they point to overlap. Note that in out case partial overlaps already suffice for references to be considered each others alias.

To explain the kind of optimisation we want to enable and why they require non-aliasing information we will discuss a concrete example right away.

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

Listing 1: Overview Example

Listing 1 defines a function `f` that takes two mutable references `x` and `y` as parameters. In the body of `f` we write the numbers `7` and `42` to the locations `x` and `y` point to and then dereference `x` and `y` and finally compare their values to constants.

We would now like to argue, that `x` and `y` do not alias each other. If this was the case, we would know that the assignment to `*y` on line 3 does not change `*x` and therefore, we could replace the read `*x` with the constant `7`. This would allow the compiler to simply return `true`, because the read `*y` could trivially be replaced with `42`.

Listing 2 shows the optimised version of the example as described in the text above.

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

Listing 2: Non-Aliasing Optimisation of Example

Rust's typesystem statically enforces non-aliasing rules that were used for the optimisation above. But those rules can be broken by using `unsafe` Rust. However, breaking those rules is generally considered as bugs by Rust developers and, as we will see, Stacked Borrows considers code were those rules are broken to be undefined behaviour (UB). Even so, we will next illustrate in the following code how our example could be broken by using `unsafe` Rust.

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

Listing 3: Unsafe Rust Code Creating Aliasing Mutable References

Listing 3 shows how two mutable references to the same location can be created and passed as two separate function arguments. This would make `x` and `y` from Listings 1 and 2 point to the same location and therefore, they would alias each other, breaking the assumption required for the shown optimisation.

Users are advised to predominantly use safe Rust and either use unsafe

Rust in concise small blocks of code or by using crates (Rust packages) 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 [2], so we have to consider unsafe Rust when we develop optimisations for Rust.

This is where Stacked Borrows comes into play, because this operational semantics also extends to `unsafe` Rust and the code in listing 3 would be considered undefined behaviour (UB). This essentially means that, if we assume the Stacked Borrows operational semantics holds for a program, we can perform the described optimisation.

1.2 Motivation

To showcase why the described optimisation is useful and not something that is already covered for Rust programs consider the following example.

```
1 fn f2(x: &mut i32, y: &mut i32) -> bool {
2     *x = 7;
3     *y = 42;
4     lib(&*x);
5     lib(&*y);
6     return *x == 7 && *y == 42;
7 }
```

Listing 4: Motivating Example

Listing 4 shows function `f2` which is very similar to `f` in listings 1 and 2. There are only two additions: in line 4 and 5 we call a black-box function `lib` that takes read-only access to `x` and `y`.

The Stacked Borrows operational semantics allow us to prove that an optimisation replacing the return statement on line 6 to return `true` is correct. This is an optimisation that is not yet performed in Rust even when running the compiler on the highest optimisation level and enabling experimental advanced optimisations.

Moreover, the compiler internals do not expose the explicit information to perform the described optimisation. To generate this information more static analysis is required. Implementing this additional static analysis is the main goal of this work. Additionally, optimisations are implemented to evaluate the quality of the generated information.

1.3 Goals

In order to address the challenges presented so far, in the beginning we defined the following goals:

- **Craft Examples:** Create code examples for the optimisation 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.
- **Gather Optimisation Requirements:** Discover what information is required to implement the desired optimisation.
- **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 optimisation.
- **Implement Optimisation:** Implement the optimisation using the information exposed by the static analysis.
- **Evaluation:** Evaluate the developed static analysis and optimisation.

1.4 Outline

The structure of this report is as follows:

- Chapter 1 introduced the problem and goals of the thesis and gives an outline of the report.
- Chapter 2 contains information about knowledge that is required to understand the presented work. This makes Stacked Borrows a special focus of the chapter.
- Chapter 3 describes our methodology: we explain idea and architecture of our work and our solution in more detail.
- Chapter 4 formalises the static analysis and the static information generated by it.
- Chapter 5 takes a closer look at the evaluation, implementation, and faced challenges.
- Chapter 6 discusses related work.
- Chapter 7 concludes this work and elaborates future work, possibilities, and opportunities.

Chapter 2

Background

This chapter contains information about knowledge that is required to understand the presented work. First we describe the programming language Rust with its aspects that are important to the thesis. Second we summarise the results of Stacked Borrows [12], on which this work is based. Last we explain some of the Rust compiler (*rustc*) internals that are required to understand our methodology.

2.1 Rust-Language

For readers unfamiliar with Rust we recommend learning about it in one of the many great free sources online. Some of those sources are: the official Rust book [13], the book called "Rust By Example" [17], and on the more humorous side the book "Learn Rust With Entirely Too Many Linked Lists" [4]. However, we try to make the core of this thesis understandable without prior knowledge of Rust by providing examples and describing Rust syntax and semantic as we go along.

2.2 Stacked Borrows

This section summarises the results of Stacked Borrows [12]. It is the work on which this thesis is based on and therefore an important part to understand the rest of this report.

Stacked Borrows [12] is an operational semantics for memory access which encodes aliasing rules for references in 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.

Stacked Borrows' rules are implemented in Miri [11], which is an interpreter for Rust that works as a dynamic analysis tool for aliasing information. Because Miri is an interpreter, executing a program is about 1000 times slower compared to running the same code compiled to a native executable. That is the main reason, why the interpreter is intended to be used for testing rather than replacing compiled Rust. Note that Miri does not create static information, but does dynamic analysis on a specific execution of a Rust application.

Stacked Borrows works by tracking which references have access to which memory locations. This access information is stored in a stack per memory location called borrow stack, hence the name Stacked Borrows. Generally every reference in the borrow stack can be used to access the referenced memory location. Before an access all entries of the borrow stack above the used reference get popped, leaving the used reference at the top of 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. Accessing a memory location by using a reference that is not on the borrow stack is undefined behaviour.

For the use in optimisations the compiler can assume an input program to conform to Stacked Borrows in every situation. A violation of the Stacked Borrows rules is undefined behaviour and in these situations the compiler is allowed to assume any behaviour.

2.3 Mid-level Intermediate Representation (MIR)

This section explains a Rust compiler (*rustc*) internal structure that is required to understand our methodology and implementation.

The Mid-level Intermediate Representation [10] [15], abbreviated as MIR, is the representation of a Rust program during the "middle" stages of compilation. It is called mid-level, because it is the representation that is used between the High-level Intermediate Representation (HIR) and LLVM [6] (the low-level presentation).

HIR is roughly an abstract syntax tree and LLVM is not a Rust specific intermediate representation. The introduction of MIR added a Rust specific low-level intermediate representation. MIR reduces Rust to a simple core: all expressions are flattened, allowing no nested expressions and all control flow statements (e.g. while, if, match, ...) are unified by creating a so called control flow graph.

A control flow graph (CFG) is a graph based representation of code. The nodes of a CFG, called basic blocks, consist a list of non-branching statements. Branching and merging of control flow is represented by edges of the CFG.

CFG representations such as MIR are useful for optimisations and so called dataflow analysis [16]. In this thesis we make use of dataflow analyses to generate static information and perform optimisation. Therefore, operating on MIR was the obvious choice for our work.

Chapter 3

Methodology

This chapter describes our methodology. We first give an overview of the scope of this thesis, second explain the idea of our work in an intuitive way, third discuss the developed static analysis in greater detail and last present the implemented optimisation.

3.1 Scope

In this section we briefly outline the scope of our work.

Our goal is to create a solution that can be run on Rust code and is sound. First this means that we want to create static analyses and optimisations that can be performed on Rust code and produce reusable information for the former and actual executable output for the latter. Second the analyses and optimisations have to be sound, meaning the generated information is correct and any performed optimisation does not change the semantics of the program it operates on. Additionally, the analyses and optimisations have to always terminate to be sound.

Our approach works with a limited set of types. Namely, our system requires reference types and more specifically mutable references. Moreover, only references that point to copy-types are accepted, this includes most prominently references to primitive types. For example the type `&mut i32` would be handled by our system. Additionally, we require the references to be parameters of the analysed function. The system still works for all of Rust while being sound, it just does not generate information and perform optimisations for types other then the ones described here.

3.2 Idea

In this section we present the core idea of our work and give an intuition how the pieces fit together.

Our plan was to design a static analysis that works as a static approximation of Stacked Borrows. The following text will discuss what kind of information our analysis has to provide. Additionally, we describe the optimisation we implemented, which we have done to evaluate that the information generated by the static analysis is relevant and non-trivial.

We will again use our motivating example for illustration:

```
1 fn f2(x: &mut i32, y: &mut i32) -> bool {
2     *x = 7;
3     *y = 42;
4     lib(&*x);
5     lib(&*y);
6     return *x == 7 && *y == 42;
7 }
```

Listing 5: Motivating Example

Recall that we want to replace the statement in line 6 with a simple `return true` resulting in eliminating two reads and one comparison. To achieve this we would like to simplify line 6 to `return 7 == 7 && 42 == 42` first and then let the compiler reduce this to `return true`. This second part is something that is already solved for Rust and we can therefore focus on the first part. To simplify this first part even more it can be seen as replacing reads like `*x` with a value. In the example this is done twice: first replacing `*x` with `7` and second replacing `*y` with `42`.

To be able to perform this optimisation the compiler needs information. More concretely the information it needs to know is:

1. The value of `*x` for a reference `x` at location l. In our example we want the compiler to know that the value of `*x` is `7` at line 2 and the value of `*y` is `42` at line 3.

2. The values of `*x` and `x` do not change between l and the location of the read. In our example we want the compiler to know that the value of `*x` did not change between lines 2 and 6 and the value of `*y` did not change between lines 3 and 6.

We combined those two properties into something we call *immutability span*. An immutability span contains the location l at which the value of `*x` is known and a set of subsequent consecutive locations for which we know that *x and x were not changed. For our example this could be visualised as shown in listing 6.

```
1 fn f2(x: &mut i32, y: &mut i32) -> bool {
2     *x = 7;
3     *y = 42;
4     lib(&*x);
5     lib(&*y);
6     return *x == 7 && *y == 42;
7 }
```

Listing 6: Motivating Example – Immutability Span Visualised

Listing 6 shows two immutability spans. The first is visualised with a green line that starts with a circle on line 2 and ends on line 6. The second is shown using a blue line that starts with a rhombus on line 3 and ends on line 6. The shapes at the start are used to be able to differentiate the lines with something other than colour and have no further purpose.

3.3 Static Analysis

In this section we describe in more detail how our static analysis works. First we look at a simplified version of the analysis that only considers straight line code, last we expand and generalise the approach to also work on more complex code that contains branches and merges of control flow.

3.3.1 Straight Line Code

In this section we expand on the intuition given about the static analysis at the beginning of the chapter.

Recall that we want to compute immutability spans that hold the information that for a set of consecutive statements the value *x did not change for a reference x. Now we will discuss how these immutability spans are computed.

To compute immutability spans we use static analysis, or more precisely we use dataflow analysis [16]. This method of static analysis works on the control flow graph (CFG) representation of the program we are analysing. We have discussed CFGs and Rust's CFG called Mid-level Intermediate Representation (MIR) in the background chapter.

When explaining how the dataflow analysis works we will always look at one *body* at a time. Bodies can be seen as the sub-graphs of the CFG that contain all nodes and edges belonging to a function. For example when looking at a Rust function `fn f`, the body of `f` is all CFG nodes and edges that result from compiling `f` to the MIR stage. We are allowed to only look at one body at a time, because Stacked Borrows, and our analysis that builds on it, enable intraprocedural reasoning.

The dataflow analysis to compute immutability spans works as follows: Find all mutable references of supported types that are passed to the current body. We will discuss which types are supported later on. Next for each reference `x` that was found, we do the following steps:

1. Find statement after which the value `*x` is known. So for example after the statement `*x = 42;` we know that `*x` has value 42. But we are not limited to compile time constants. So for the assignment `*x = get_user_input();` the analysis also considers the value of `*x` to be known. We use assignments here as an example, but the approach is not restricted assignments.
2. For each statement from step 1 find all consecutive statements for which the following conditions hold:
   - `*x` is not modified,
   - `x` is not changed to point to a different location, and
   - the statement does not give away mutable access to `*x`. For example the following statement would violate this last condition: `write_to(&mut *x);`
3. Construct immutability span out of the statement found in step 1 and the statements found in step 2.

We will illustrate these steps in our motivating example in listing 7. The resulting immutability spans from that example are:

- *i1* for `x`: Starting at line 5 and ending at line 18.
- *i2* for `y`: Starting at line 10 and ending at line 18.

Notice that the two resulting immutability spans are overlapping. This is usual for different references. For the same reference we will make sure that its immutability spans do not overlap each other, by merging those overlapping immutability spans together.

3.3.2 Branches and Loops

The approach discussed so far works well for functions in which we can just look at the statements as a consecutive list. But we already know, that the data structure we look at is a directed graph which is allowed to have branches and cycles. So we need to generalise our approach to work with CFGs.

```
// Found mutable references x and y.
fn f2(x: &mut i32, y: &mut i32) -> bool {
    // *x is known after the following statement.
    // Immutability Span i1 starts here.
    *x = 7;

    // *y is known after the following statement.
    // Immutability Span i2 starts here.
    // i1: All conditions are fulfilled => add to i1
    *y = 42;

    // i1: All conditions are fulfilled => add to i1
    // i2: All conditions are fulfilled => add to i2
    lib(&*x);

    // i1: All conditions are fulfilled => add to i1
    // i2: All conditions are fulfilled => add to i2
    return *x == 7 && *y == 42;
}
```

Listing 7: Example for Finding Immutability Span

First we will briefly look into immutability spans again and how this affects them. So far we have defined the set of statements in the immutability span to consist of consecutive statements but we kept it vague and did not describe what consecutive means in this context. For blocks (nodes of the CFG) it is simple: inside of a block statements are in an absolute order and statements are consecutive with statements directly before and after themselves according to that order.

However, to be able to provide non-trivial instances of immutability spans, we require a definition of consecutiveness that can span across block boundaries. In those boundary cases a statement can have more than two consecutive statements. Namely, the first statement of a block b is consecutive to the last statements of all blocks that are predecessors to b in the CFG. Analogously, the last statement of a block b_2 is consecutive to the first statements of every block that is a successor of b_2.

For simplification we will look at branches and merges in the CFG and not consider more complex structures such as cycles as a whole. Every branching or merging of control flow could potentially be a part of a cycle and this option has to be considered in the analysis. Inside of blocks (nodes of the CFG) we keep the described approach from the previous section. But we change the overall order of execution and define how the analysis works for transitions between blocks (i.e. what happens on branches and merges).

```
dirty := copy(body.blocks)
analysis_information :=
    { (index, bottom) for index in body.statements.indices }
while |dirty| > 0:
    block := dirty.pop()
    for i := 0..|block.statements|:
        statement := block.statements[i]
        old_information := analysis_information[statement.index]
        new_information := analyse(statement)

        if i == 0:  for predecessor in block.predecessors:
            new_information :=
                join(new_information, predecessor.information)

        analysis_information[statement.index] := new_information

        if i == |block.statements| - 1:
            block.information := new_information
            if old_information != new_information:
                dirty := union(dirty, block.successors)
```

Listing 8: Pseudocode that Describes Dataflow Analysis

This generalised approach works as illustrated by the pseudocode in listing 8. This listing shows a general dataflow algorithm, which was not created as part of this work, but is used here to explain how dataflow analysis works. The algorithm loops over blocks until it converges to a fixpoint. This fixpoint essentially means that no more changes to analysis information would occur to any block b at this point if we run analysis again for block b.

Here we will briefly discuss the algorithm in the pseudocode in listing 8: it loops over blocks until there are no "dirty" blocks left and we process one block at a time. A block b is part of the `dirty` set, if b was not yet processed or if at least one of b's predecessors changed its analysis information since b was last processed. Processing a block works by looking at the individual statements of the block in order and doing following steps:

1. Analyse the current statement.
2. If it was the first statement of the block, then use the `join` function to combine the analysis information of predecessor blocks with the analysis information of the first statement of the current block. We will discuss how the `join` function is implemented later on.
3. If it was the last statement of the block, check if the analysis information changed. If it changed add all successors of the current block to

the `dirty` set.

With some implementations of `join` and `analyse` the code could loop endlessly. To make sure it is an algorithm and always terminates for any input, some restrictions for how `join` and `analyse` work have to be enforced. We will go into the details of those restrictions in the formalisation chapter.

Next we will look into what kind of analysis data we use in the discussed algorithm to be able to find immutability spans using it. The analysis data we store per statement is: a mapping $V \to \bot|I|\top$, where V is the set of all local variables of the current body and I is the set of immutability spans. So a statement can be part or not part of an immutability span for every local variable. $\bot$ means the statement was not processed yet and $\top$ means the analysis cannot conclude if the statement is part of an immutability span for the given variable. $\top$ and $\bot$ will be further discussed in the formalisation chapter.

The function `join` has to merge different analysis information coming from different blocks in a meaningful way. Using the definition from above join has the following type: $join : AxA \to A$ where $A = (V \to \bot|I|\top)$ `join` for our analysis is simply defined as:

$$join(A,B)(v) := \begin{cases} \bot, & \text{if } A(v) = B(v) = \bot \\ i \in I, & \text{if } A(v) = B(v) = i \\ i \in I, & \text{if } A(v) = i \land B(v) = \bot \\ i \in I, & \text{if } B(v) = i \land A(v) = \bot \\ \top, & \text{otherwise} \end{cases}$$

3.4 Optimisation

In this section we describe in more detail how the optimisation works.

The optimisation tries to use the generated immutability spans to perform optimisations. More specifically, we want to eliminate reads from memory. We will again use our motivating example in listing 9 to discuss how the optimisation works. Recall that we want to replace the reads in line 6 so that the line is simplified to `return 7 == 7 && 42 == 42`. The compiler already has optimisations in place that will further reduce that to `return true`.

Using the immutability spans performing the optimisation is not too involved: at the point of the write we store the right hand side of the assignment in a temporary local variable. Then for all reads inside of the immutability span we replace the read with the local variable. For our example this would look as shown in listing 10 using two immutability spans, one for each reference `x` and `y`.

```rust
fn f2(x: &mut i32, y: &mut i32) -> bool {
    *x = 7;
    *y = 42;
    lib(&*x);
    lib(&*y);
    return *x == 7 && *y == 42;
}
```

Listing 9: Motivating Example

```rust
fn f2(x: &mut i32, y: &mut i32) -> bool {
    let local_x = 7;
    *x = local_x;
    let local_y = 42;
    *y = local_y;
    lib(&*x);
    lib(&*y);
    return local_x == 7 && local_y == 42;
}
```

Listing 10: Motivating Example Optimised

Note that we did not replace the reads in the return statement with the numbers directly but with a read-only local. For the compiler and further optimisations this is almost equivalent, as a simple constant propagation can replace the variable in the return statement with the correct constants. However, using those read-only locals makes our approach way more flexible, as we can also do replacements as shown in listings 11 and 12.

```rust
fn f3_original(x: &mut i32) -> i32 {
    *x = read_input();
    lib(&*x);
    return *x;
}
```

Listing 11: f3 Original Input

```rust
fn f3_optimised(x: &mut i32) -> i32 {
    let local_x = read_input();
    *x = local_x;
    lib(&*x);
    return local_x;
}
```

Listing 12: f3 Original Optimised

Listing 11 shows the original unmodified function, while 12 shows the result after our optimisation was performed on it. Note that this time we did not write a constant to `*x` but the result of a function call. If we replaced any read from `*x` in the immutability span with that function call we would change the semantics of the program, as function calls can have side-effects. By using the local variable we can avoid changing the semantics of the program and are allowed to perform the optimisation as it is shown in listing 12.

Chapter 4

Formalisation

In this chapter we formalise the immutability spans and the static analysis that is used to find them in given Rust programs.

In this chapter we assume that the reader has a more profound knowledge of this work's background. Namely, we assume a familiarity with Rust [13] and its terminology and importantly with rustc internals such as MIR [10]. Additionally, we assume background knowledge about static analysis. Concretely, we will discuss dataflow analysis and assume the reader is familiar with terms like "lattice", "transfer function", and "fixpoint". As a source to learn more about dataflow analysis we recommend the book named "Static Program Analysis" [16] which available online for free.

4.1 Immutability Span

In the methodology chapter we discussed what we call immutability spans and we provided an intuitive understanding of them. In this section we describe what an immutability span is in a more formal way.

We define an immutability span as $I = (r, l, S)$ with

- r the local for which the immutability span holds,
- l the start location of the immutability span, and
- S a set of locations.

l and every $s \in S$ are locations which are defined as a pair (b, i) where b is the block index and i is the statement index inside that block. So locations point to specific statements in the MIR of the target body. We use $STMT(x)$ to denote the statement at location x in the target body.

An immutability span $I = (r, l, S)$ fulfills the following properties:

- r, l and every $s \in S$ belong to the same MIR body which we call the target body of this immutability span.
- r is a mutable reference that contains a type implementing the `Copy` trait.
- At runtime after $STMT(l)$ has been executed `*r` has a value v. For every $s \in S$ the value of `*r` is v just before $STMT(s)$ is being executed. Note that v does not have to be known at compile time.
- The locations in S are consecutive. The locations in S are consecutive if thew following two conditions are fulfilled:

$$\begin{aligned} \forall(b_1, i_1), (b_2, i_2) \in S \ . \ b_1 = b_2 \wedge i_1 = i_2 + 2 \implies \\ \exists(b_t, i_t) \in S \ . \ b_t = b_1 \wedge i_1 < i_t < i_2 \end{aligned} \tag{4.1}$$

$$\begin{aligned} &\forall(b_1, i_1), (b_2, i_2) \in S \ . \ b_2 \in SUCCESSORS(b_1) \implies \\ &LAST\_LOCATION(b_1) \in S \wedge (b_2, 0) \in S \end{aligned} \tag{4.2}$$

  Where $SUCCESSORS(b)$ denotes the set of indices of all successor blocks of b in the target MIR body and $LAST\_LOCATION(b)$ denotes the location (b, i) for which holds that there exists no location (b, i') with $i' > i$.

4.2 Static Analysis

This section gives a more formal definition of the static analyses that we implemented for this thesis.

So far we only discussed static analysis as a combined unit. However, we actually implemented the static analysis as two separate dataflow analyses, with the second depending on the output of the first one. Splitting the analyses in that way has the following advantages:

- Formalisation and reasoning are simplified. We can formalise the analyses separately and reason about correctness individually rather than on one big combined system.
- The implementation becomes more structured and therefore easier to read, understand and maintain.
- The first analysis can be used separately and independently from the second analysis. This improves re-usability of our work.

- The first analysis called ”Top of Borrow Stack” is in a general form that allows usage of a specialised fixpoint iteration algorithm which significantly reduces runtime and memory usage.[1]

Disadvantages of splitting the static analysis into two analyses are:

- It requires extra effort to design and implement, since a clear interface has to be defined and data cannot be freely shifted between the two analyses.

The two dataflow analyses are named ”Top of Borrow Stack” and ”Find Immutability Spans” and they are described in the following sections.

4.2.1 Top of Borrow Stack Analysis

This section describes the ”Top of Borrow Stack” analysis. Recall that the Stacked Borrows operational semantics tracks a borrow stack per memory location. The ”Top of Borrow Stack” is an approximation of said borrow stack. Concretely, for each local reference r of supported type our analysis collects the information for which locations we can guarantee that r is on top of every borrow stack for the place it points to.

We can better understand this by looking at the output of this analysis: the analysis creates a set $T : R \times L$, where R is the set of locals and L the set of locations in the target MIR body. Every $(r, l) \in T$ means:

1. r is a local,
2. r is a function parameter for the target body,
3. r is of supported type, concretely r's type is mutable reference to a `Copy` type,
4. r is at location l guaranteed to be on top of every borrow stack for the place r points to.

In contrast, for a local r of supported type and a location, $(r, l) \notin T$ means we do not know if local r at location l is or is not on top of the borrow stacks for the place r points to. So the two options are: guaranteed to be on top and unknown.

Next we would like to clarify two points. First we discuss what exactly we mean with ”the place r points to”. Stacked Borrows uses for borrow stacks memory locations in byte resolution. For example a 16-bit integer would have two borrow stacks, one for the first byte and the second for the other byte, and those borrow stacks could be completely different from each

[1] Read more about this in the evaluation chapter, where we talk about the rustc internal `GenKillAnalysis`.

other. For our analysis we only consider a simplified version and only track a single borrow stack per type. To keep this simplification correct, $(r, l) \in T$ only holds for a local r and location l, if we can guarantee that r is at location l for each byte of the place r points to at the top of this bytes borrow stack.

Second we discuss what we mean with a local r being in top of a borrow stack. As we have seen this analysis only tracks mutable references. This is the case because we are interested in aliasing information for and more precisely we would like to provide a guarantee that the value $*r$ for a reference r was not modified for a set of consecutive locations. So with that in mind we can more precisely define which types of borrow stacks provide the guarantees we require. For the sake of this analysis we consider r to be on top of a borrow stack for a byte-sized memory location if:

1. *Unique(r)* is the top-most element of the borrow stack,
2. the borrow stack contains *Unique(r)* and above this entry only contains items of permission type *SharedRO* above it.

Unique and *SharedRO* are parts of Stacked Borrows definition of the borrow stack. What the above conditions mean is that only the local r has write permission to the value $*r$, while other references might have or might have had read permission to the value.

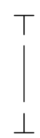

Figure 4.1: Top of Borrow Stack – Lattice

Finally we take a brief look at the lattice used for the "Top of Borrow Stack" analysis. Figure 4.1 shows the developed lattice. $\bot$ stands for "on top of the borrow stack" while $\top$ means "unknown". This is because of how the join operation is defined on a lattice: $\bot \sqcup \top = \top$. For example for a block with two predecessors where we receive "unknown" on one edge and "on top of the borrow stack" on the other, we want the analysis to continue in the "unknown" state to ensure no incorrect information is propagated.

4.2.2 Find Immutability Spans Analysis

This section describes the "Find Immutability Spans" analysis. This analysis uses the output of the "Top of Borrow Stack" analysis as input and generates immutability spans as output. You can find formal descriptions of immutability spans and the "Top of Borrow Stack" analysis in earlier sections of this chapter.

Figure 4.2 shows the lattice used in this analysis. It is more complex then the lattice we have looked at before in the previous section. The elements of the lattice have the following meanings:

- $\bot$ stands for "uninitialised state". It is set for every pair of local and location before the analysis is run.
- $\top$ stands again for "unknown" similar to the previously discussed lattice.
- $Span(I_n)$ for a pair (r, l) means local r at location l is part of immutability span I_n.

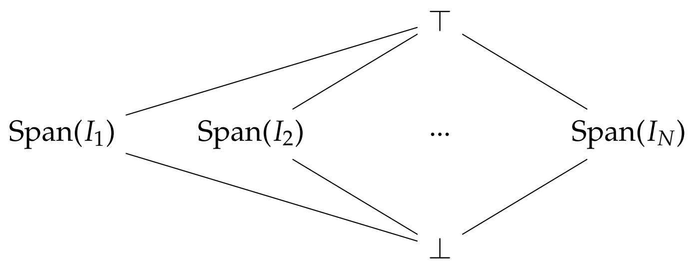

Figure 4.2: Find Immutability Spans – Lattice

As an example we could look at what $Span(I_1) \sqcup Span(I_{42}) = \top$ means for a pair (r, l): we look at a basic block where one predecessor has the state $Span(I_1)$ (r is in the predecessor part of immutability span I_1) and the other $Span(I_2)$ (r is in the other predecessor part of immutability span I_{42}) for local r. On joining that information we know that r is either part of I_1 or I_{42} at location l. Because we want to output a single immutability span for a pair of local and location, the join operation is defined to return $\top$ (it is "unknown" which immutability span(s) r is part of) in this case.

After running the analysis we get a set $O : R \times L \times S$, where R is the set of locals and L the set of locations for the target body and S is the set of possible lattice states ($S = \{\bot, \top\} \cup \{Span(I_i) | i \in \mathbb{N}\}$). We use this set O to create a collection of immutability spans as they were defined earlier in this section. To do so we collect every $(r, l, s) \in O$ which has identical local r and identical s with $s \notin \{\bot, \top\}$ in a set. This set is the set of locations of the immutability span, with exception of the single location which has no preceding location that is part of the set. This special location is used as the start location of the immutability span and not part of the set of locations.

Chapter 5

Evaluation

This chapter describes our evaluation, faced challenges and implementation.

In table 5.1 we link to our code repositories and provide information on which versions we used for evaluation.

Static Analyses and Optimisation	`https://github.com/janispeyer/rustc_alias/`
Changes to Rustc	`https://github.com/janispeyer/rust`
Rustc commit our changes are based on	1e926f0
Commits used for evaluation	d95ed85 (rustc_alias) and f29fff9 (rust)

Table 5.1: Our Repositories and Important Commits

5.1 Testing

This section gives insight into how we tested the implementation.

5.1.1 Manual Testing

In this section we describe how we manually checked the different outputs when running our optimisation compared to when the Rust compiler was run without our changes. We will show how our optimisation is able to remove instructions that the unmodified compiler will not and therefore produce a more runtime efficient output. We will demonstrate this on a version of our motivating example shown in listing 13.

```rust
#[inline(never)]
fn lib(x: &u32) {
    // Prevent the LLVM inferring that the argument is unused
    println!("{}", x);
}

#[inline(never)]
fn mid(x: &mut u32, y: &mut u32) -> bool {
    *x = 7;
    *y = 42;
    lib(x);
    lib(y);
    return *x == 7 && *y == 42;
}

fn main() {
    let mut x = 1;
    let mut y = 2;
    println!("{}", mid(&mut x, &mut y));
}
```

Listing 13: Motivating Example Used for Manual Testing

This listing additionally defines a main function that calls our example function and also defines `lib`. This additional functions have to be defined to make the example build. Moreover, main has to call our example function, so that our function does not get eliminated by a dead-code elimination optimisation. For similar reasons we print `x` in the function `lib` to make sure neither the parameter nor the call to `lib` get eliminated by optimisations.

Next we discuss the created assembly code presented in listing 14. On the left hand side of the listing we show what was produced by the unmodified compiler and on the right hand side what was produced by the compiler with out optimisations. Both use the Rust code from listing 13 as input. Note that we only discuss a shortened version of the assembly code here which additionally has renamed labels for better readability. The full version of the output can be found in appendix A.

In listing 14 we can clearly see that the version using our optimisation is significantly shorter. The left hand side of the listing contains instructions for comparing `*x` and `*y` to 7 and 42 starting at line 17. All those instructions for those comparisons are completely optimised away in the version on the right hand side. This improvement was performed and enabled by our optimisation.

```
move_past_shared_borrow__mid:
.seh_proc move_past_shared_borrow__mid
        push    rsi
        .seh_pushreg rsi
        push    rdi
        .seh_pushreg rdi
        sub     rsp, 40
        .seh_stackalloc 40
        .seh_endprologue
        mov     rsi, rdx
        mov     rdi, rcx
        mov     dword ptr [rcx], 7
        mov     dword ptr [rdx], 42
        call    move_past_shared_borrow__lib
        mov     rcx, rsi
        call    move_past_shared_borrow__lib
        mov     eax, dword ptr [rdi]
        xor     eax, 7
        mov     ecx, dword ptr [rsi]
        xor     ecx, 42
        or ecx, eax
        sete    al
        add     rsp, 40
        pop     rdi
        pop     rsi
        ret
```

```
move_past_shared_borrow__mid:
.seh_proc move_past_shared_borrow__mid
        push    rsi
        .seh_pushreg rsi
        sub     rsp, 32
        .seh_stackalloc 32
        .seh_endprologue
        mov     rsi, rdx
        mov     dword ptr [rcx], 7
        mov     dword ptr [rdx], 42
        call    move_past_shared_borrow__lib
        mov     rcx, rsi
        add     rsp, 32
        pop     rsi
        jmp     move_past_shared_borrow__lib
```

Listing 14: Assembly Output from Compiler Without (left) and With Our Optimisation (right)

```
rustc +nightly -Zmir-emit-retag --emit asm -C llvm-args=-x86-asm-syntax=intel -C opt-level=3 tests\ui\eliminate_reads\move_past_shared_borrow.rs -o original.s

cargo run -- -Zmir-emit-retag --emit asm -C llvm-args=-x86-asm-syntax=intel -C opt-level=3 tests\ui\eliminate_reads\move_past_shared_borrow.rs -o optimised.s
```

Listing 15: CommandLine

Figure 15 shows the two command-lines that were used to generate the presented assembly outputs. The upper one was used to create the unoptimised version, while the lower was used to create the optimised version.

Noteworthy is that we used `opt-level=3` to turn on the highest level of optimisations and that we used the nightly version of Rust. Rust nightly includes additional experimental optimisations which are not part of the stable version. This shows that our optimisation can perform improvements that were previously not done in Rust.

Moreover we also tested Rust nightly with the additional flag `-Zunsound-mir-opts` to turn on experimental features of the new constant propagation [14]. We write more about the new constant propagation in the related work chapter. Using this flag resulted in the same unoptimised output as the one on the left hand side of listing 14.

Another thing that can be seen nicely in listing 15 is that the crate containing our additions can be directly used as if they were rustc: we use `cargo run` to run our crate and pass rustc command-line arguments directly to it. This is made possible by using `rustc_driver` which will be discussed in the implementation section later on.

5.1.2 Automated Testing

To test the analyses and optimisation we created an automated test suite that can be run when changes were made to check, if the change broke any part of the system. The tests can be run using the command `cargo test` which is commonly used for automated tests in Rust projects. We use the crate (Rust package) `compiletest_rs` [5] to automate our tests.

Every `compiletest_rs` test consists of two files: one containing some Rust code and the other the expected analysis output and expected optimisation. The former can be any Rust program. Additionally, it can contain compile flags and options in the form of comments to control how the test is run. The latter contains the analysis information for every line of MIR (Mid-level Intermediate Representation) and the MIR before and after performing optimisations.

If a change leads to a different analysis output or if it was optimised differently the test would me flagged as fail and could then be resolved by either fixing the code if there was a bug or adjusting the test to match the change in the analyses or optimisations. There is also a flag called `bless` that can be used to generate the output files for all tests instead of testing the new output against the files that are already present.

This approach also does not only test the correctness of the output but also tests for things like crashes and indicates how much time our system takes to run. If running the tests would for example show a large increase in runtime we would quickly notice when running the tests and would be able to resolve the issue as soon as it occurs.

Optimisations Performed (Primitive Types)	
Test File	What is being tested?
`eliminate_reads.rs`	Basic test with assignments to `*x` and reads to `*x` that get eliminated.
`immutability_span_tuple_assignment.rs`	Tests if more complicated assignments like tuple destructuring get handled correctly in our system.
`immutability_span.rs`	Tests if immutabilty spans get constructed correctly for more complex CFGs with branching and merging of control flow.
`move_local.rs`	Test to check that assignments of more complex expressions instead of a constant, such as a read from another variable, get handled correctly.
`move_past_shared_borrow.rs`	Test that checks if the analysis can correctly handle read elimination when a shared borrow of the target reference was created between the assignment and the read. This is also the motivating example from the introduction.
`regain_of_top_of_stack.rs`	Tests that the analysis correctly detects that an assignment to `*x` means that `x` is on top of the borrow stack for the target location, even if it was not before the assignment.
`repeat_write.rs`	Test if a chain of assignments to `*x` gets handled correctly. More specifically it checks, if one assignment that is part of two immutability spans (e.g. `*x = *x;`) is handled correctly.

Table 5.2: Automated tests and what they test: Optimisations Performed (Primitive Types)

There are a total of 25 files for automated tests, where each file tests a different feature or syntax. Tables 5.2, 5.3, 5.4, and 5.5 show a list of all test files and what they test.

5.1.3 Official Rust Test Suite

We also experimented with testing our system using the `rustc` test suite. On the commit of `rustc` we based our changes on (without modifications) the test suite passes 26'106 tests successfully before encountering a block where every tests fails (181 failed tests).

When running the test suite with our modifications (analyses and optimisation) all tests that passed without modifications passed again with them.

Optimisation Not Allowed (Primitive Types)	
Test File	What is being tested?
`call_return.rs`	Test to check that no optimisation is performed for `*x`, if `x` is assigned to between the assignment to and read from `*x`.
`cast_to_pointer.rs`	Tests that no optimisation is performed for `*x`, if `x` is cast to a pointer between the assignment to and read from `*x`.
`inline_asm.rs`	Tests that no optimisation is performed that needs information that spans across `asm` (Assembly) blocks.
`interior_mutability.rs`	Tests that references to types with interior mutability are not involved in any of our optimisations.
`loop_with_borrow_in_body.rs`	Tests that mutable borrows of `*x` inside a loop prevent optimisations from being performed if they need information that spans across the loop. Additionally, we checked here that the analysis does not consider `x` to be part of any immutability span in the statements inside the loop but before the borrow of `*x`, because that would be an error.
`mem_replace.rs`	Test to check that using `std::mem::replace` on `x` does correctly prevent optimisations if it is located between the assignment to and read from `*x`.
`nested_reference.rs`	Test to make sure nested references are not part of optimisations.
`reborrow_by_function_call.rs` and `reborrow.rs`	Contain tests that checks that creating a mutable borrow of `*x` correctly prevents optimisations that need information that span across that mutable borrow.

Table 5.3: Automated tests and what they test: Optimisation Not Allowed (Primitive Types)

Optimisations Performed (Non-Primitive Types)	
Test File	What is being tested?
custom_tuple.rs	Tests that optimisations are performed for custom tuple types that implement the Copy trait.
custom_types.rs	Tests that optimisations are performed for custom struct and enum types that implement the Copy trait.
enum.rs	Tests that optimisations are performed for primitive enum types (tag only enums) that implement the Copy trait.
option.rs	Tests that optimisations are performed for the Option<T> type.
partial_access.rs	Tests that reads from custom struct and tuple types get optimised even if only a single field is accessed instead of a full read of all the data. (e.g. reading x.0 instead of *x) The optimisation is only performed if the custom types implement the Copy trait.
shared_reference.rs	Shared references implement the Copy trait. This test checks that optimisations are performed for mutable references to shared references. (e.g. for x: &mut &i32)
tuple.rs	Tests that optimisations are performed for native tuple types. (i.e. for tuple types that are not custom tuple types.) Note that optimisations for native tuple types are not yet performed as this would require some refactoring. This is discussed in the section about the implementation.

Table 5.4: Automated tests and what they test: Optimisations Performed (Non-Primitive Types)

Optimisation Not Allowed (Non-Primitive Types)	
Test File	What is being tested?
internal_reference_custom_type.rs	Tests that mutable borrows of fields of custom types prevent optimisations if they need information that spans across it.
internal_reference_tuple.rs	Tests that mutable borrows of tuple elements prevent optimisations if they need information that spans across it.

Table 5.5: Automated tests and what they test: Optimisation Not Allowed (Non-Primitive Types)

Meaning all 26'106 tests passed again when running our analyses and optimisations. We appended a shortened version of the output of running the test suite including our analyses and optimisations in the appendix B.

Running the tests with an optimisation that intentionally introduces errors if applied made the test suite stop in the first test block were it found errors. This shows that our code is actually run and would be able to introduce errors that can be found by the test suite.

We were not able to create a profound certainty that our work is correct from these tests. However, we could draw some conclusions: we learned that our code can run on a large amount of Rust code without crashing. Additionally, we could confirm that at least some optimisations are being performed in the test suite and that the test suite does not find any error with the performed optimisations.

5.2 Implementation

This section explains the details of our implementation and the challenges we faced.

We first like to mention that the static analyses that are used to find immutability spans are designed to be reused on their own. More concretely, one of our design goals was to allow third parties to reuse the analysis without having to also run our optimisations code. The idea behind this is that software, like the formal verification tool Prusti [1] or IDE extensions that improve the developer experience, could reuse our analysis information.

5.2.1 Architecture

In this section we will briefly discuss the architecture. This will serve as a guide through the following chapters which contain class diagrams for each part of the architecture.

Figure 5.1: Architecture

Figure 5.1 shows an overview of the architecture. The boxes symbolise modules and the arrows the flow of data between the modules. There are three modules: two containing a static analysis each and one containing the implemented optimisation.

For each of these modules there is a section including its own class diagram, because a combined class diagram would be too large and visually cluttered. Moreover, the diagrams do not include life-times and have some parameters and functions stripped away to improve clarity. To make the different diagrams form a bigger picture and to be able to see their relation to one another there are some duplicated elements. Namely every class diagram contains the `Alias` struct which runs the static analyses and optimisation in its `run_pass` function.

In addition to the three modules discussed so far, there is also a section about the compiler injection point. This is a change we did directly to the compiler, which allows us to add our static analyses and optimisation as a so-called MIR-pass into the Rust compiler. There was previously no way to hook into the compiler like that to add additional MIR-passes.

5.2.2 Static Analysis: Top of Borrow Stack

Figure 5.2 shows the class diagram for the static analysis "Top of Borrow Stack".

At the top of the diagram resides the previously discussed `Alias` struct. It implements the `MirPass` trait from the rustc internals, which is used to represent an unit of code that analyses and modifies a MIR-body. For example a optimisation such as constant propagation can be implemented as a `MirPass`. `Alias` and `MirPass` are part of every of the following class diagrams to give an orientation how the different diagrams fit together.

`Alias` uses a struct called `PrintBodyVisitor` which is used to print the MIR to the standard output. This is used for the testing framework to print the CFG before and after our optimisation is performed and then compare that printed output against the expected output stored in a file on disk. `PrintBodyVisitor` implements the `Visitor` trait from the rustc internals, which is an implementation of the visitor design pattern [8] for MIR.

The static analyses use the built-in dataflow library of the Rust compiler. This library already contains a function to iterate dataflow analyses to their fixpoints, over useful ways to access analysis results, and generally provide handy features and a framework to build dataflow analyses.

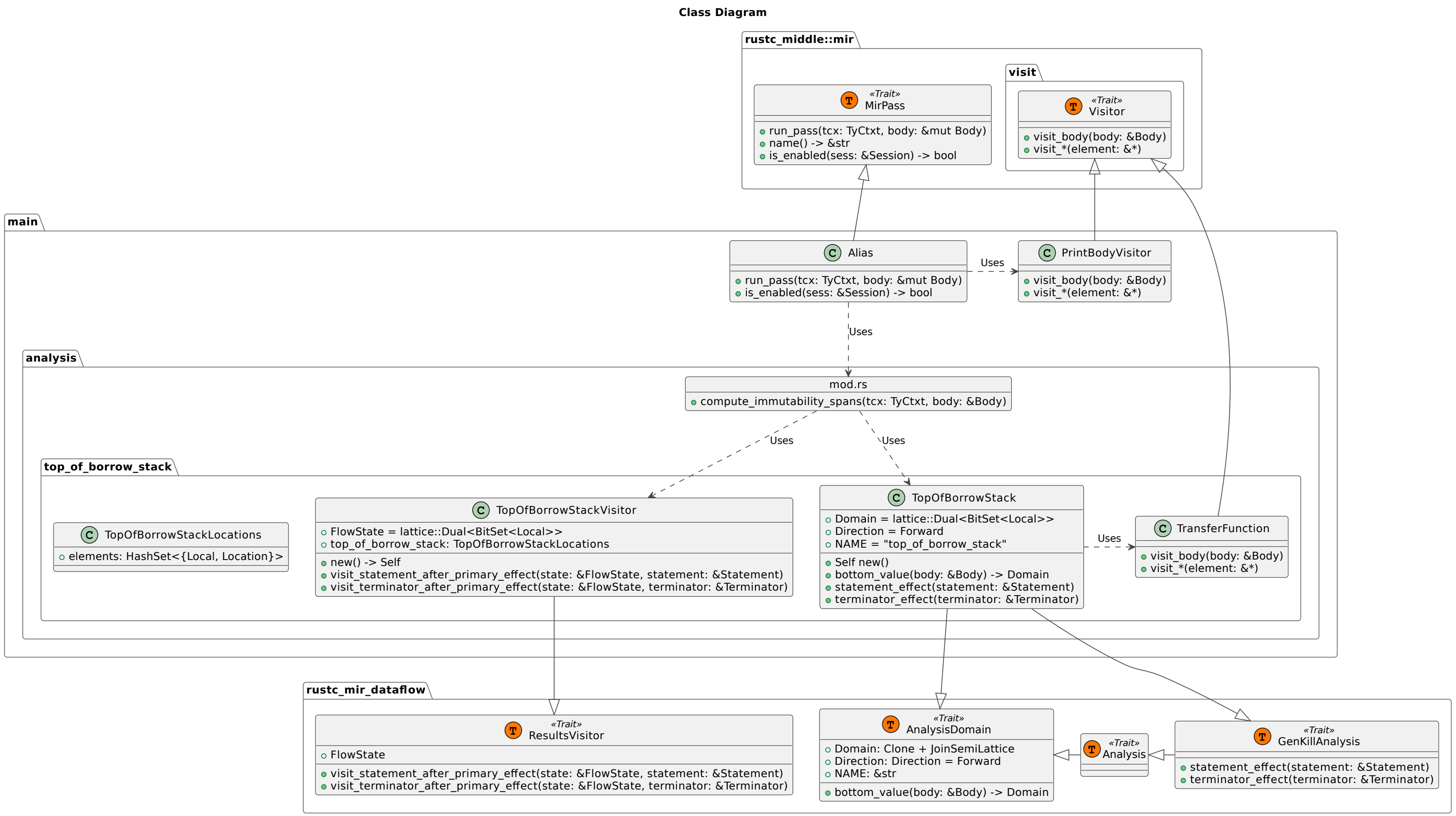

Figure 5.2: Class Diagram – Static Analysis: Top of Borrow Stack

In figure 5.2 we can see how the dataflow library is used to implement the "Top of Borrow Stack" static analysis. The easiest way to discuss the diagram in that figure is to follow the execution path.

1. The `run_pass` function of `Alias` calls the function `compute_immutability_spans`. There the two static analyses get called after each other, feeding the result of the first analysis as input to the second one.

2. To compute the "Top of Borrow Stack" analysis, the `TopOfBorrowStack` struct is used. This struct implements the traits `AnalysisDomain` and `Analysis` of the rustc internals. Those traits require that `TopOfBorrowStack` defines a dataflow analysis and in turn allow us to pass `TopOfBorrowStack` to the dataflow library to be iterated to the fixpoint on a MIR-body.

   Note that `TopOfBorrowStack` does not implement the `Analysis` directly, but it implements the `GenKillAnalysis` trait which is a subtrait of `Analysis` and therefore also implements `Analysis` indirectly. `GenKillAnalysis` is a specialised dataflow analysis that works on a minimal non-trivial lattice which only contains $\top$ and $\bot$. The dataflow library uses a specialised algorithm to iterate to the fixpoint of `GenKillAnalysis` which reduces runtime and memory usage.

3. `TopOfBorrowStack` uses our `TransferFunction` struct to define what exactly has to happen for every element of a MIR-body. It also implements the `Visitor` trait, which we have already discussed in an earlier paragraph, to do so.

4. After `TopOfBorrowStack` was used to iterate to the fixpoint, the `TopOfBorrowStackVisitor` is used to collect the results in our data structure `TopOfBorrowStackLocations`. `TopOfBorrowStackVisitor` implements the `ResultVisitor` trait, which is an implementation of the visitor design pattern [8] for dataflow analysis results. `TopOfBorrowStackLocations` contains for each `Location` every `Local` that is considered "top of borrow stack" according to our static analysis.

5. The resulting `TopOfBorrowStackLocations` instance is then passed to the "Find Immutability Spans" static analysis, which is discussed in the following section.

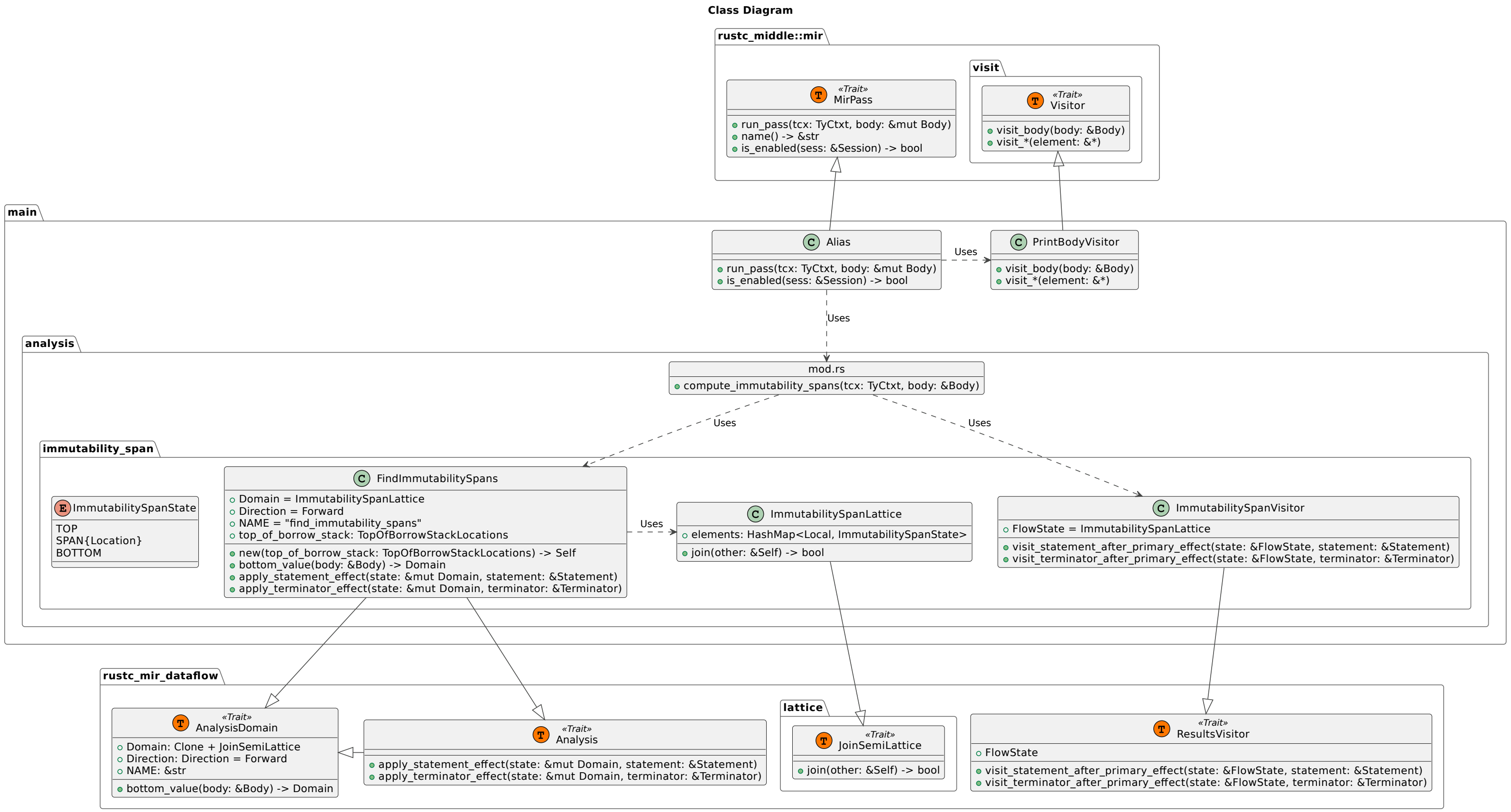

Figure 5.3: Class Diagram – Static Analysis: Find Immutability Spans

5.2.3 Static Analysis: Find Immutability Spans

Figure 5.3 shows the class diagram for the static analysis ”Find Immutability Spans”. The structre of this dataflow analysis is analogue to the structure of the ”Top of Borrow Stack” analysis we discussed previously. We will discuss the structure again by following the execution path, but will keep it brief and focus on the differences to the previously discussed dataflow analysis.

1. Recall that the function `compute_immutability_spans` runs the ”Top of Borrow Stack” analysis. The result of this analysis is then passed to the ”Find Immutability Spans”, which the function runs next.
2. To compute the ”Find Immutability Spans” analysis, the `FindImmutabilitySpans` struct is used. Similarly to the previous analysis this struct implements the traits `AnalysisDomain` and `Analysis`. However, here the `Analysis` trait is implemented directly, because the used lattice is more complex and does therefore not fit the requirements for `GenKillAnalysis`.
3. The lattice used by `FindImmutabilitySpans` is defined in the struct `ImmutabilitySpanLattice`. To be used as lattice it has to implement the `JoinSemiLattice` trait and define the `join` function which joins two values of the lattice. To represent lattice values the `ImmutabilitySpanState` enum is used.
4. After `FindImmutabilitySpans` was used to iterate to the fixpoint, the `ImmutabilitySpanVisitor` struct is used to create the immutability spans from the dataflow analysis result. `compute_immutability_spans` uses this visitor to create the immutability spans and return them to `Alias` which passes the immutability spans to the optimisation.

   Note that the immutability spans can also be created without running the optimisations. To skip optimisations the function `compute_immutability_spans` can be called directly instead of using the `Alias` struct which runs the static analyses and optimisations both.

5.2.4 Optimisation

Figure 5.4 shows the class diagram for the optimisation ”Eliminate Reads”.

Using immutability spans performing this optimisation only requires very little complexity to be performed and we see that reflected in the minimal class diagram. Here again we will discuss the diagram by following the execution path.

1. After creating the immutability spans for a body `Alias` calls the `eliminate_reads` function. The immutability spans are passed as argument to this function.

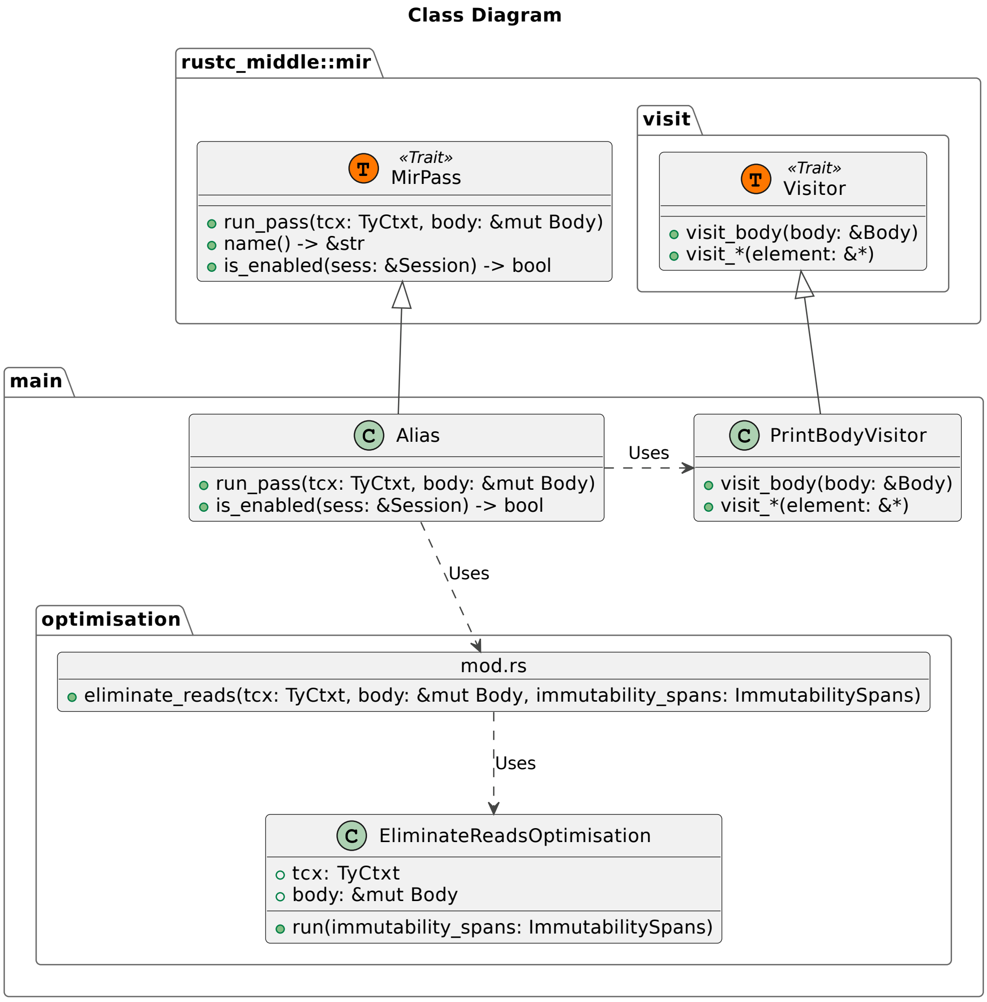

Figure 5.4: Class Diagram – Optimisation

2. The `eliminate_reads` function creates the `EliminatereadsOptimisation` struct and performs the optimisations by calling the `run` function on this struct.

5.2.5 Compiler Injection

Figure 5.5 shows the class diagram for the compiler injection point.

All of our functions and structs we have shown so far are part of our own crate which does not reside inside the Rust compiler and is not part of a rust-lang repository [7]. We link our crate to rustc and use `rustc_driver` [9] to run the compiler. This allows our crate to be run as if it was rustc with all the usual command-line and configuration options.

The difference to running rustc directly is that we hook into the compiler to

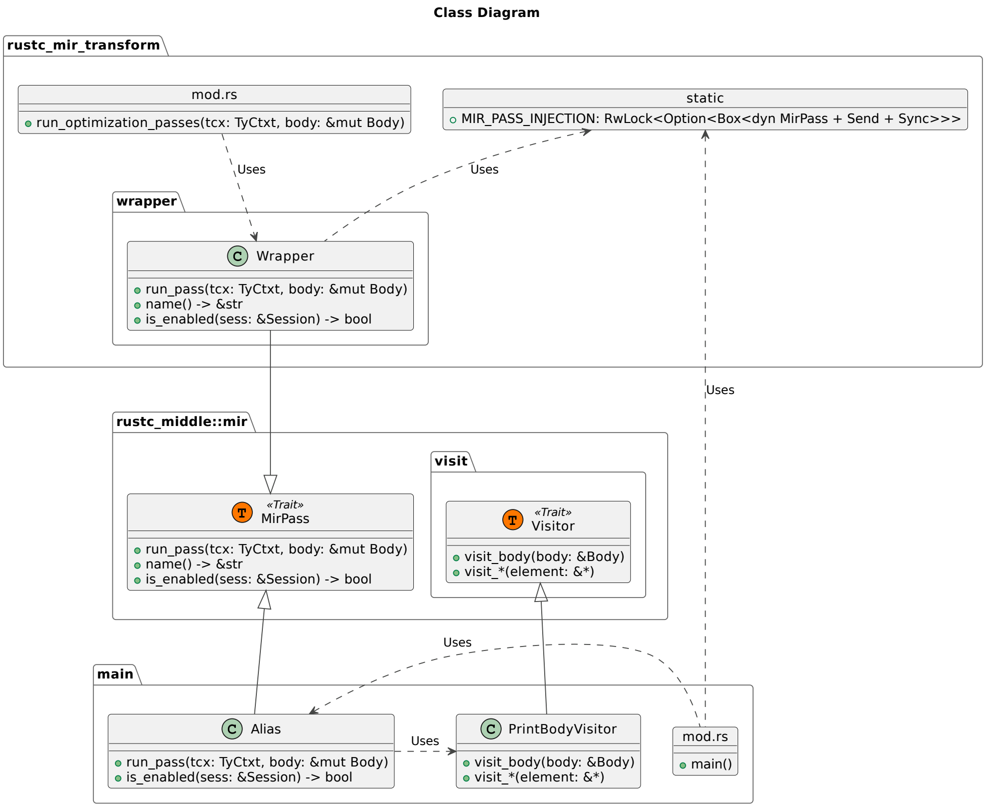

Figure 5.5: Class Diagram – Compiler Injection

run our static analyses and optimisation during the compilers optimisation phase. To achieve this we had to modify rustc to allow injection of our static analyses and optimisation into rustc when using `rustc_driver`.

This injection point is added in the form of a static location in rustc to which a `MirPass` can be written. Recall that `Alias` which runs our static analyses and optimisation implements the `MirPass` trait and can therefore be written to this injection point. The `main` function of our crate does just that: it writes `Alias` to the injection point and then runs rustc using `rustc_driver`.

To run the `MirPass` that is stored at the injection point we use the wrapper design pattern (also known as adapter pattern [8]). This pattern is implemented by the `Wrapper` struct which too implements the `MirPass` trait. The `Wrapper` is then statically tied into the `run_optimization_passes` function of rustc and for every function called on `Wrapper` it forwards the function call to the `MirPass` stored at the injection point.

Chapter 6

Related Work

This chapter discusses work that is related to this thesis.

6.1 Miri

Miri [11] is an interpreter for Rust's mid-level intermediate representation. It is a tool that was mainly developed for testing and can be used to find memory-safety violations that occur because of bugs in unsafe Rust code. For example it can detect out-of-bounds memory accesses and use-after-free bugs.

Miri is relevant to this work, because Stacked Borrows checks are implemented in it and Miri can detect undefined behaviour in Rust programs that is caused by violating Stacked Borrows rules.

Miri performs all checks dynamically at runtime while interpreting the target Rust program. This is a fundamental difference to our approach, in which we perform all checks and modifications statically at compile time. Additionally, Miri tries to keep the target as-is, whereas we explicitly set out to perform optimisations.

6.2 LLVM

LLVM [6] is a compiler infrastructure that provides a unified backend for compilers such as clang (modern C/C++ compiler) and prominently the Rust compiler. Rust uses the LLVM backend for code generation and to optimise programs. LLVM is one of the most effective backends for optimisations [3].

Therefore, few optimisations have to be performed in Rust directly and most can be delegated to the LLVM backend. However, there are opportunities to leverage Rust's unique ownership based typesystem to implement optimisations that cannot be performed by LLVM. The optimisation developed in this thesis is such an optimisation which cannot be performed by LLVM, because LLVM's typesystem and model do not allow for such optimisations.

Note that a Rust user could force this kind of optimisation on a case-by-case basis by using `assume` intrinsic statements. These statements can be used to provide additional information to LLVM that allow it to perform the desired optimisations. However, it is inherently unsafe to use these `assume` intrinsics, as they can very easily introduce bugs and lead to memory-safety violations. Their usage is comparable to directly injecting assembly instructions into Rust code: the desired output is achieved but the approach is error prone and could easily lead to memory-safety violations.

6.3 Constant Propagation

Recently a new constant propagation optimisation was added to the Rust compiler [14]. Similar to this thesis it leverages the MIR dataflow library internals of rustc. Moreover, parts of it relies on properties of Stacked Borrows. These parts are gated behind the command-line flag `-Zunsound-mir-opts` because Stacked Borrows should still be considered an experiment until it is officially integrated into Rust's specification.

However, the constant propagation optimisation performs a different set of optimisations than the optimisation that we developed in this thesis. Our optimisation is alias based and operates on references where it tries to eliminate reads. Concretely, it can eliminate those reads for values that are not constant.

Chapter 7

Conclusion

This chapter concludes our work and elaborates future work, possibilities, and opportunities.

In this thesis we created static dataflow analyses that leverage the Stacked Borrows operational semantics [12]. Additionally, we created optimisations to evaluate the quality of the static information generated by the analyses. We tested the correctness of the analyses and optimisation with a suite of tests that we manually created. Moreover, we run the analyses and optimisations on the official rustc test suite which contains thousands of test programs.

We have shown how the Stacked Borrows operational semantics can be used to generate static information which is relevant and non-trivial. We expose a program interface that allows third parties to generate the static information we call immutability spans. Importantly, the interface allows the static analyses to be run without also performing our optimisation.

7.1 Future Work

This section describes possible future work and opportunities.

7.1.1 Tree Borrows

Very recently a variation of Stacked Borrows called Tree Borrows was introduced [18]. Tree Borrows does not use a borrow stack per memory location but instead uses a tree. The main purpose of the change is to reduce the amount of undefined behaviour and, by doing so, supporting Rust applications that previously contained undefined behaviour when using Stacked Borrows.

This work could be improved by evaluating if our approach is still valid when using Tree Borrows instead of Stacked Borrows. This would make our work applicable to more Rust programs.

7.1.2 Increase Precision

The precision of our analysis could be further improved in several ways. We could add support for shared references and generate static information that include them. Another addition might be to support locals that are not function parameters.

7.1.3 Additional Optimisations

Our work could be leveraged to implement additional optimisations. This might be done without adjusting the static analysis. However, the static analysis could also be extended to match the needs of new optimisations.

7.1.4 Extended Evaluation

The evaluation of our static analyses and optimisation could be extended to achieve further certainty that our approach is correct or find bugs and inconsistencies otherwise. An opportunity to extend the evaluation would be to test our approach on published crates (Rust packages) or to run public benchmarks and test suites with our static analyses and optimisation enabled.

7.1.5 Benchmark Optimisation

We mainly created the optimisation to evaluate if the generated static information is non-trivial and relevant and therefore did not consider the runtime improvements that result from performing our optimisation. However, it could be useful to analyse improvements to further examine the claim that our approach is beneficial.

To measure runtime improvements official benchmarks may be used or a custom benchmark could be created. Additionally or alternatively, the effectiveness of the optimisation might be measured on existing Rust code.

7.1.6 Tooling Integration

Last we would like to suggest the integration of our static analyses into existing tooling. Especially projects like Prusti [1] would benefit from static analysis that leverages Stacked Borrows. By doing so the precision of formal verification could be improved and unsafe Rust might be better supported.

Appendix A

Rustc Test Suite Output

In this appendix we add the full assembly code that was created by the unmodified compiler in section A.1 and the compiler that includes our optimisation in listing A.2. You can find a discussion of the parts that are relevant to this thesis in the evaluation chapter under section 5.1.1.

A.1 Assembly Code Output from Unmodified Compiler

```
	.text
	.def	 @feat.00;
	.scl	3;
	.type	0;
	.endef
	.globl	@feat.00
.set @feat.00, 0
	.intel_syntax noprefix
	.file	"move_past_shared_borrow.d5f7624b-cgu.0"
	.def	 _ZN3std10sys_common9backtrace28__rust_begin_short_backtrace17hf84105e2d4a26754E;
	.scl	3;
	.type	32;
	.endef
	.section	.text,"xr",one_only,_ZN3std10sys_common9backtrace28__rust_begin_short_backtrace17hf84105e2d4a26754E
	.p2align	4, 0x90
_ZN3std10sys_common9backtrace28__rust_begin_short_backtrace17hf84105e2d4a26754E:
```

```
.seh_proc _ZN3std10sys_common9backtrace28__rust_begin_short_backtrace17hf84105e2d4a26754E
        sub     rsp, 40
        .seh_stackalloc 40
        .seh_endprologue
        call    rcx
        #APP
        #NO_APP
        nop
        add     rsp, 40
        ret
        .seh_endproc

        .def     _ZN3std2rt10lang_start17h5617b6987bc9a47cE;
        .scl    2;
        .type   32;
        .endef
        .section        .text,"xr",one_only,_ZN3std2rt10lang_start17h5617b6987bc9a47cE
        .globl  _ZN3std2rt10lang_start17h5617b6987bc9a47cE
        .p2align        4, 0x90
_ZN3std2rt10lang_start17h5617b6987bc9a47cE:
.seh_proc _ZN3std2rt10lang_start17h5617b6987bc9a47cE
        sub     rsp, 56
        .seh_stackalloc 56
        .seh_endprologue
        mov     rax, r8
        mov     r8, rdx
        mov     qword ptr [rsp + 48], rcx
        mov     byte ptr [rsp + 32], r9b
        lea     rdx, [rip + __unnamed_1]
        lea     rcx, [rsp + 48]
        mov     r9, rax
        call     _ZN3std2rt19lang_start_internal17hfa9601e856a0d3d7E
        nop
        add     rsp, 56
        ret
        .seh_endproc

```

```
        .def        _ZN3std2rt10lang_start28_$u7b$$u7b$closure$u7d$$u7d$17h03bd544d0f913043E;
        .scl        3;
        .type        32;
        .endef
        .section        .text,"xr",one_only,_ZN3std2rt10lang_start28_$u7b$$u7b$closure$u7d$$u7d$17h03bd544d0f913043E
        .p2align        4, 0x90
_ZN3std2rt10lang_start28_$u7b$$u7b$closure$u7d$$u7d$17h03bd544d0f913043E:
.seh_proc _ZN3std2rt10lang_start28_$u7b$$u7b$closure$u7d$$u7d$17h03bd544d0f913043E
        sub        rsp, 40
        .seh_stackalloc 40
        .seh_endprologue
        mov        rcx, qword ptr [rcx]
        call        _ZN3std10sys_common9backtrace28__rust_begin_short_backtrace17hf84105e2d4a26754E
        xor        eax, eax
        add        rsp, 40
        ret
        .seh_endproc

        .def        _ZN44_$LT$$RF$T$u20$as$u20$core..fmt..Display$GT$3fmt17h38733ca35cc1a335E;
        .scl        3;
        .type        32;
        .endef
        .section        .text,"xr",one_only,_ZN44_$LT$$RF$T$u20$as$u20$core..fmt..Display$GT$3fmt17h38733ca35cc1a335E
        .p2align        4, 0x90
_ZN44_$LT$$RF$T$u20$as$u20$core..fmt..Display$GT$3fmt17h38733ca35cc1a335E:
        mov        rcx, qword ptr [rcx]
        jmp        _ZN4core3fmt3num3imp52_$LT$impl$u20$core..fmt..Display$u20$for$u20$u32$GT$3fmt17h80da57ee0e46922aE

        .def        _ZN4core3ops8function6FnOnce40call_once$u7b$$u7b$vtable.shim$u7d$$u7d$17h7bff86a46a4de388E;
        .scl        3;
```

```
	.type	32;
	.endef
	.section	.text,"xr",one_only,_ZN4core3ops8function6FnOnce40call_once$u7b$$u7b$vtable.shim$u7d$$u7d$17h7bff86a46a4de388E
	.p2align	4, 0x90
_ZN4core3ops8function6FnOnce40call_once$u7b$$u7b$vtable.shim$u7d$$u7d$17h7bff86a46a4de388E:
.seh_proc _ZN4core3ops8function6FnOnce40call_once$u7b$$u7b$vtable.shim$u7d$$u7d$17h7bff86a46a4de388E
	sub	rsp, 40
	.seh_stackalloc 40
	.seh_endprologue
	mov	rcx, qword ptr [rcx]
	call	_ZN3std10sys_common9backtrace28__rust_begin_short_backtrace17hf84105e2d4a26754E
	xor	eax, eax
	add	rsp, 40
	ret
	.seh_endproc

	.def	_ZN4core3ptr85drop_in_place$LT$std..rt..lang_start$LT$$LP$$RP$$GT$..$u7b$$u7b$closure$u7d$$u7d$$GT$17hacdddc16b040c770E;
	.scl	3;
	.type	32;
	.endef
	.section	.text,"xr",one_only,_ZN4core3ptr85drop_in_place$LT$std..rt..lang_start$LT$$LP$$RP$$GT$..$u7b$$u7b$closure$u7d$$u7d$$GT$17hacdddc16b040c770E
	.p2align	4, 0x90
_ZN4core3ptr85drop_in_place$LT$std..rt..lang_start$LT$$LP$$RP$$GT$..$u7b$$u7b$closure$u7d$$u7d$$GT$17hacdddc16b040c770E:
	ret

	.def	_ZN23move_past_shared_borrow3lib17h7ff9aeb2eb57efe8E;
	.scl	3;
	.type	32;
	.endef
```

```
        .section        .text,"xr",one_only,_ZN23move_past_shared_borrow3lib17h7ff9aeb2eb57efe8E
        .p2align        4, 0x90
_ZN23move_past_shared_borrow3lib17h7ff9aeb2eb57efe8E:
.seh_proc _ZN23move_past_shared_borrow3lib17h7ff9aeb2eb57efe8E
        sub         rsp, 104
        .seh_stackalloc 104
        .seh_endprologue
        mov         qword ptr [rsp + 32], rcx
        lea         rax, [rsp + 32]
        mov         qword ptr [rsp + 40], rax
        lea         rax, [rip + _ZN44_$LT$$RF$T$u20$as$u20$core..fmt..Display$GT$3fmt17h38733ca35cc1a335E]
        mov         qword ptr [rsp + 48], rax
        lea         rax, [rip + __unnamed_2]
        mov         qword ptr [rsp + 56], rax
        mov         qword ptr [rsp + 64], 2
        mov         qword ptr [rsp + 72], 0
        lea         rax, [rsp + 40]
        mov         qword ptr [rsp + 88], rax
        mov         qword ptr [rsp + 96], 1
        lea         rcx, [rsp + 56]
        call         _ZN3std2io5stdio6_print17h9b42b865ab0fe9c8E
        nop
        add         rsp, 104
        ret
        .seh_endproc

        .def         _ZN23move_past_shared_borrow3mid17h0db60ad6d5244510E;
        .scl         3;
        .type         32;
        .endef
        .section        .text,"xr",one_only,_ZN23move_past_shared_borrow3mid17h0db60ad6d5244510E
        .p2align        4, 0x90
_ZN23move_past_shared_borrow3mid17h0db60ad6d5244510E:
.seh_proc _ZN23move_past_shared_borrow3mid17h0db60ad6d5244510E
        push         rsi
```

```
        .seh_pushreg rsi
        push         rdi
        .seh_pushreg rdi
        sub          rsp, 40
        .seh_stackalloc 40
        .seh_endprologue
        mov          rsi, rdx
        mov          rdi, rcx
        mov          dword ptr [rcx], 7
        mov          dword ptr [rdx], 42
        call         _ZN23move_past_shared_borrow3lib17h7ff9aeb2eb57efe8E
        mov          rcx, rsi
        call         _ZN23move_past_shared_borrow3lib17h7ff9aeb2eb57efe8E
        mov          eax, dword ptr [rdi]
        xor          eax, 7
        mov          ecx, dword ptr [rsi]
        xor          ecx, 42
        or           ecx, eax
        sete         al
        add          rsp, 40
        pop          rdi
        pop          rsi
        ret
        .seh_endproc

        .def         _ZN23move_past_shared_borrow4main17hd472ba83054a845eE;
        .scl         3;
        .type        32;
        .endef
        .section     .text,"xr",one_only,_ZN23move_past_shared_borrow4main17hd472ba83054a845eE
        .p2align     4, 0x90
_ZN23move_past_shared_borrow4main17hd472ba83054a845eE:
.seh_proc _ZN23move_past_shared_borrow4main17hd472ba83054a845eE
        sub          rsp, 120
        .seh_stackalloc 120
        .seh_endprologue
        mov          dword ptr [rsp + 48], 1
        mov          dword ptr [rsp + 52], 2
```

```
        lea         rcx, [rsp + 48]
        lea         rdx, [rsp + 52]
        call        _ZN23move_past_shared_borrow3mid17h0db60ad6d5244510E
        mov         byte ptr [rsp + 47], al
        lea         rax, [rsp + 47]
        mov         qword ptr [rsp + 56], rax
        lea         rax, [rip + _ZN43_$LT$bool$u20$as$u20$core..fmt..Display$GT$3fmt17hc1c69044d432a1aaE]
        mov         qword ptr [rsp + 64], rax
        lea         rax, [rip + __unnamed_2]
        mov         qword ptr [rsp + 72], rax
        mov         qword ptr [rsp + 80], 2
        mov         qword ptr [rsp + 88], 0
        lea         rax, [rsp + 56]
        mov         qword ptr [rsp + 104], rax
        mov         qword ptr [rsp + 112], 1
        lea         rcx, [rsp + 72]
        call        _ZN3std2io5stdio6_print17h9b42b865ab0fe9c8E
        nop
        add         rsp, 120
        ret
        .seh_endproc

        .def         main;
        .scl         2;
        .type         32;
        .endef
        .section         .text,"xr",one_only,main
        .globl        main
        .p2align        4, 0x90
main:
.seh_proc main
        sub         rsp, 56
        .seh_stackalloc 56
        .seh_endprologue
        mov         r9, rdx
        movsxd         r8, ecx
        lea         rax, [rip + _ZN23move_past_shared_borrow4main17hd472ba83054a845eE]
        mov         qword ptr [rsp + 48], rax
```

```
        mov     byte ptr [rsp + 32], 2
        lea     rdx, [rip + __unnamed_1]
        lea     rcx, [rsp + 48]
        call     _ZN3std2rt19lang_start_internal17hfa9601e856a0d3d7E
        nop
        add     rsp, 56
        ret
        .seh_endproc

        .section        .rdata,"dr",one_only,__unnamed_1
        .p2align        3
__unnamed_1:
        .quad   _ZN4core3ptr85drop_in_place$LT$std..rt..lang_start$LT$$LP$$RP$$GT$..$u7b$$u7b$closure$u7d$$u7d$$GT$17hacdddc16b040c770E
        .asciz  "\b\000\000\000\000\000\000\000\b\000\000\000\000\000\000"
        .quad   _ZN4core3ops8function6FnOnce40call_once$u7b$$u7b$vtable.shim$u7d$$u7d$17h7bff86a46a4de388E
        .quad   _ZN3std2rt10lang_start28_$u7b$$u7b$closure$u7d$$u7d$17h03bd544d0f913043E
        .quad   _ZN3std2rt10lang_start28_$u7b$$u7b$closure$u7d$$u7d$17h03bd544d0f913043E

        .section        .rdata,"dr",one_only,__unnamed_3
        .p2align        3
__unnamed_3:

        .section        .rdata,"dr",one_only,__unnamed_4
__unnamed_4:
        .byte   10

        .section        .rdata,"dr",one_only,__unnamed_2
        .p2align        3
__unnamed_2:
        .quad   __unnamed_3
        .zero   8
        .quad   __unnamed_4
        .asciz   "\001\000\000\000\000\000\000"
```

A.2 Assembly Code Output from Compiler With Our Optimisation

```
	.text
	.def	 @feat.00;
	.scl	3;
	.type	0;
	.endef
	.globl	@feat.00
.set @feat.00, 0
	.intel_syntax noprefix
	.file	"move_past_shared_borrow.4cc28df5-cgu.0"
	.def	 _ZN3std10sys_common9backtrace28__rust_begin_short_backtrace17h07b15b46742f7420E;
	.scl	3;
	.type	32;
	.endef
	.section	.text,"xr",one_only,_ZN3std10sys_common9backtrace28__rust_begin_short_backtrace17h07b15b46742f7420E
	.p2align	4, 0x90
_ZN3std10sys_common9backtrace28__rust_begin_short_backtrace17h07b15b46742f7420E:
.seh_proc _ZN3std10sys_common9backtrace28__rust_begin_short_backtrace17h07b15b46742f7420E
	sub	rsp, 40
	.seh_stackalloc 40
	.seh_endprologue
	call	rcx
	#APP
	#NO_APP
	nop
	add	rsp, 40
	ret
	.seh_endproc

	.def	 _ZN3std2rt10lang_start17h9fb9706d27b0a4a5E;
	.scl	2;
	.type	32;
	.endef
```

```
	.section	.text,"xr",one_only,_ZN3std2rt10lang_start17h9fb9706d27b0a4a5E
	.globl	_ZN3std2rt10lang_start17h9fb9706d27b0a4a5E
	.p2align	4, 0x90
_ZN3std2rt10lang_start17h9fb9706d27b0a4a5E:
.seh_proc _ZN3std2rt10lang_start17h9fb9706d27b0a4a5E
	sub	rsp, 56
	.seh_stackalloc 56
	.seh_endprologue
	mov	rax, r8
	mov	r8, rdx
	mov	qword ptr [rsp + 48], rcx
	mov	byte ptr [rsp + 32], r9b
	lea	rdx, [rip + __unnamed_1]
	lea	rcx, [rsp + 48]
	mov	r9, rax
	call	_ZN3std2rt19lang_start_internal17hae3a6cd3dffcbabdE
	nop
	add	rsp, 56
	ret
	.seh_endproc

	.def	_ZN3std2rt10lang_start28_$u7b$$u7b$closure$u7d$$u7d$17h9f945db4ed42001eE;
	.scl	3;
	.type	32;
	.endef
	.section	.text,"xr",one_only,_ZN3std2rt10lang_start28_$u7b$$u7b$closure$u7d$$u7d$17h9f945db4ed42001eE
	.p2align	4, 0x90
_ZN3std2rt10lang_start28_$u7b$$u7b$closure$u7d$$u7d$17h9f945db4ed42001eE:
.seh_proc _ZN3std2rt10lang_start28_$u7b$$u7b$closure$u7d$$u7d$17h9f945db4ed42001eE
	sub	rsp, 40
	.seh_stackalloc 40
	.seh_endprologue
	mov	rcx, qword ptr [rcx]
	call	_ZN3std10sys_common9backtrace28__rust_begin_short_backtrace17h07b15b46742f7420E
```

```
        xor         eax, eax
        add         rsp, 40
        ret
        .seh_endproc

        .def         _ZN44_$LT$$RF$T$u20$as$u20$core..fmt..Display$GT$3fmt17h56f17affb4d02bd2E;
        .scl         3;
        .type         32;
        .endef
        .section         .text,"xr",one_only,_ZN44_$LT$$RF$T$u20$as$u20$core..fmt..Display$GT$3fmt17h56f17affb4d02bd2E
        .p2align         4, 0x90
_ZN44_$LT$$RF$T$u20$as$u20$core..fmt..Display$GT$3fmt17h56f17affb4d02bd2E:
        mov         rcx, qword ptr [rcx]
        jmp         _ZN4core3fmt3num3imp52_$LT$impl$u20$core..fmt..Display$u20$for$u20$u32$GT$3fmt17h621a09d895289ab4E

        .def         _ZN4core3ops8function6FnOnce40call_once$u7b$$u7b$vtable.shim$u7d$$u7d$17hc631771b1b35eaa8E;
        .scl         3;
        .type         32;
        .endef
        .section         .text,"xr",one_only,_ZN4core3ops8function6FnOnce40call_once$u7b$$u7b$vtable.shim$u7d$$u7d$17hc631771b1b35eaa8E
        .p2align         4, 0x90
_ZN4core3ops8function6FnOnce40call_once$u7b$$u7b$vtable.shim$u7d$$u7d$17hc631771b1b35eaa8E:
.seh_proc _ZN4core3ops8function6FnOnce40call_once$u7b$$u7b$vtable.shim$u7d$$u7d$17hc631771b1b35eaa8E
        sub         rsp, 40
        .seh_stackalloc 40
        .seh_endprologue
        mov         rcx, qword ptr [rcx]
        call         _ZN3std10sys_common9backtrace28__rust_begin_short_backtrace17h07b15b46742f7420E
        xor         eax, eax
        add         rsp, 40
        ret
```

```
	.seh_endproc

	.def	 _ZN4core3ptr85drop_in_place$LT$std..rt..lang_start$LT$$LP$$RP$$GT$..$u7b$$u7b$closure$u7d$$u7d$$GT$17hc6a66a411ac5e918E;
	.scl	3;
	.type	32;
	.endef
	.section	.text,"xr",one_only,_ZN4core3ptr85drop_in_place$LT$std..rt..lang_start$LT$$LP$$RP$$GT$..$u7b$$u7b$closure$u7d$$u7d$$GT$17hc6a66a411ac5e918E
	.p2align	4, 0x90
_ZN4core3ptr85drop_in_place$LT$std..rt..lang_start$LT$$LP$$RP$$GT$..$u7b$$u7b$closure$u7d$$u7d$$GT$17hc6a66a411ac5e918E:
	ret

	.def	 _ZN23move_past_shared_borrow3lib17hb6fd11e7c5dafdb5E;
	.scl	3;
	.type	32;
	.endef
	.section	.text,"xr",one_only,_ZN23move_past_shared_borrow3lib17hb6fd11e7c5dafdb5E
	.p2align	4, 0x90
_ZN23move_past_shared_borrow3lib17hb6fd11e7c5dafdb5E:
.seh_proc _ZN23move_past_shared_borrow3lib17hb6fd11e7c5dafdb5E
	sub	rsp, 104
	.seh_stackalloc 104
	.seh_endprologue
	mov	qword ptr [rsp + 32], rcx
	lea	rax, [rsp + 32]
	mov	qword ptr [rsp + 40], rax
	lea	rax, [rip + _ZN44_$LT$$RF$T$u20$as$u20$core..fmt..Display$GT$3fmt17h56f17affb4d02bd2E]
	mov	qword ptr [rsp + 48], rax
	lea	rax, [rip + __unnamed_2]
	mov	qword ptr [rsp + 56], rax
	mov	qword ptr [rsp + 64], 2
	mov	qword ptr [rsp + 72], 0
```

```
        lea         rax, [rsp + 40]
        mov         qword ptr [rsp + 88], rax
        mov         qword ptr [rsp + 96], 1
        lea         rcx, [rsp + 56]
        call         _ZN3std2io5stdio6_print17hcf4262332c593811E
        nop
        add         rsp, 104
        ret
        .seh_endproc

        .def         _ZN23move_past_shared_borrow3mid17h39686d38a1877841E;
        .scl         3;
        .type         32;
        .endef
        .section         .text,"xr",one_only,_ZN23move_past_shared_borrow3mid17h39686d38a1877841E
        .p2align         4, 0x90
_ZN23move_past_shared_borrow3mid17h39686d38a1877841E:
.seh_proc _ZN23move_past_shared_borrow3mid17h39686d38a1877841E
        push         rsi
        .seh_pushreg rsi
        sub         rsp, 32
        .seh_stackalloc 32
        .seh_endprologue
        mov         rsi, rdx
        mov         dword ptr [rcx], 7
        mov         dword ptr [rdx], 42
        call         _ZN23move_past_shared_borrow3lib17hb6fd11e7c5dafdb5E
        mov         rcx, rsi
        add         rsp, 32
        pop         rsi
        jmp         _ZN23move_past_shared_borrow3lib17hb6fd11e7c5dafdb5E
        .seh_endproc

        .def         _ZN23move_past_shared_borrow4main17hd4f9d74737d1aea0E;
        .scl         3;
        .type         32;
```

```
	.endef
	.section	.text,"xr",one_only,_ZN23move_past_shared_borrow4main17hd4f9d74737d1aea0E
	.p2align	4, 0x90
_ZN23move_past_shared_borrow4main17hd4f9d74737d1aea0E:
.seh_proc _ZN23move_past_shared_borrow4main17hd4f9d74737d1aea0E
	sub	rsp, 120
	.seh_stackalloc 120
	.seh_endprologue
	mov	dword ptr [rsp + 48], 1
	mov	dword ptr [rsp + 52], 2
	lea	rcx, [rsp + 48]
	lea	rdx, [rsp + 52]
	call	_ZN23move_past_shared_borrow3mid17h39686d38a1877841E
	mov	byte ptr [rsp + 47], 1
	lea	rax, [rsp + 47]
	mov	qword ptr [rsp + 56], rax
	lea	rax, [rip + _ZN43_$LT$bool$u20$as$u20$core..fmt..Display$GT$3fmt17hba3811e99eb3064bE]
	mov	qword ptr [rsp + 64], rax
	lea	rax, [rip + __unnamed_2]
	mov	qword ptr [rsp + 72], rax
	mov	qword ptr [rsp + 80], 2
	mov	qword ptr [rsp + 88], 0
	lea	rax, [rsp + 56]
	mov	qword ptr [rsp + 104], rax
	mov	qword ptr [rsp + 112], 1
	lea	rcx, [rsp + 72]
	call	_ZN3std2io5stdio6_print17hcf4262332c593811E
	nop
	add	rsp, 120
	ret
	.seh_endproc

	.def	 main;
	.scl	 2;
	.type	 32;
	.endef
	.section	.text,"xr",one_only,main
```

```
        .globl      main
        .p2align        4, 0x90
main:
.seh_proc main
        sub         rsp, 56
        .seh_stackalloc 56
        .seh_endprologue
        mov         r9, rdx
        movsxd          r8, ecx
        lea         rax, [rip + _ZN23move_past_shared_borrow4main17hd4f9d74737d1aea0E]
        mov         qword ptr [rsp + 48], rax
        mov         byte ptr [rsp + 32], 2
        lea         rdx, [rip + __unnamed_1]
        lea         rcx, [rsp + 48]
        call         _ZN3std2rt19lang_start_internal17hae3a6cd3dffcbabdE
        nop
        add         rsp, 56
        ret
        .seh_endproc

        .section        .rdata,"dr",one_only,__unnamed_1
        .p2align        3
__unnamed_1:
        .quad         _ZN4core3ptr85drop_in_place$LT$std..rt..lang_start$LT$$LP$$RP$$GT$..$u7b$$u7b$closure$u7d$$u7d$$GT$17hc6a66a411ac5e918E
        .asciz         "\b\000\000\000\000\000\000\000\b\000\000\000\000\000\000"
        .quad         _ZN4core3ops8function6FnOnce40call_once$u7b$$u7b$vtable.shim$u7d$$u7d$17hc631771b1b35eaa8E
        .quad         _ZN3std2rt10lang_start28_$u7b$$u7b$closure$u7d$$u7d$17h9f945db4ed42001eE
        .quad         _ZN3std2rt10lang_start28_$u7b$$u7b$closure$u7d$$u7d$17h9f945db4ed42001eE

        .section        .rdata,"dr",one_only,__unnamed_3
        .p2align        3
__unnamed_3:

        .section        .rdata,"dr",one_only,__unnamed_4
__unnamed_4:
```

```
        .byte       10

        .section        .rdata,"dr",one_only,__unnamed_2
        .p2align        3
__unnamed_2:
        .quad       __unnamed_3
        .zero       8
        .quad       __unnamed_4
        .asciz       "\001\000\000\000\000\000\000"
```

Appendix B

Rustc Test Suite Output

The following contains a shortened output of the rustc test suite when run with the modified compiler containing our optimisations. Here we show that thousands of test programs were run and passed while our static analyses and optimisation were being run on all these tests. Note that the failed tests at the end of the output are not caused by our additions but also appear when running the tests on the unmodified version of the compiler. Read more about these tests in the evaluation chapter under section 5.1.3.

```
Check compiletest suite=ui mode=ui
test result: ok. 13495 passed; 0 failed; 157 ignored; 0
↪  measured; 0 filtered out; finished in 575.17s

Check compiletest suite=run-pass-valgrind mode=run-pass-valgrind
test result: ok. 17 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 7.92s

Check compiletest suite=mir-opt mode=mir-opt
test result: ok. 183 passed; 0 failed; 5 ignored; 0 measured; 0
↪  filtered out; finished in 22.62s

Check compiletest suite=codegen mode=codegen
test result: ok. 314 passed; 0 failed; 61 ignored; 0 measured; 0
↪  filtered out; finished in 7.71s

Check compiletest suite=codegen-units mode=codegen-units
test result: ok. 39 passed; 0 failed; 3 ignored; 0 measured; 0
↪  filtered out; finished in 7.08s

Check compiletest suite=assembly mode=assembly
```

```
test result: ok. 120 passed; 0 failed; 26 ignored; 0 measured; 0
↪  filtered out; finished in 2.10s

Check compiletest suite=incremental mode=incremental
test result: ok. 155 passed; 0 failed; 3 ignored; 0 measured; 0
↪  filtered out; finished in 48.62s

Check compiletest suite=ui-fulldeps mode=ui
test result: ok. 48 passed; 0 failed; 23 ignored; 0 measured; 0
↪  filtered out; finished in 9.00s

Check compiletest suite=rustdoc mode=rustdoc
test result: ok. 559 passed; 0 failed; 6 ignored; 0 measured; 0
↪  filtered out; finished in 88.84s

Check compiletest suite=pretty mode=pretty
test result: ok. 71 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 1.87s

Testing ["alloc", "core", "panic_abort", "panic_unwind",
↪  "proc_macro", "std", "test", "unwind"] stage1
test result: ok. 373 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.11s
test result: ok. 651 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.91s
test result: ok. 448 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.40s
test result: ok. 1493 passed; 0 failed; 2 ignored; 0 measured; 0
↪  filtered out; finished in 1.00s
test result: ok. 408 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.03s
test result: ok. 930 passed; 0 failed; 4 ignored; 0 measured; 0
↪  filtered out; finished in 12.32s
test result: ok. 7 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.00s
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.00s
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.11s
test result: ok. 12 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.00s
test result: ok. 58 passed; 0 failed; 1 ignored; 0 measured; 0
↪  filtered out; finished in 0.48s
```

```
test result: ok. 654 passed; 0 failed; 4 ignored; 0 measured; 0
↪  filtered out; finished in 51.15s
test result: ok. 3912 passed; 0 failed; 36 ignored; 0 measured;
↪  0 filtered out; finished in 222.93s
test result: ok. 1081 passed; 0 failed; 20 ignored; 0 measured;
↪  0 filtered out; finished in 51.60s

Testing ["rustc-main", "rustc_apfloat", "rustc_arena",
↪  "rustc_ast", "rustc_ast_lowering", "rustc_ast_passes",
↪  "rustc_ast_pretty", "rustc_attr", "rustc_borrowck",
↪  "rustc_builtin_macros", "rustc_codegen_llvm",
↪  "rustc_codegen_ssa", "rustc_const_eval",
↪  "rustc_data_structures", "rustc_driver",
↪  "rustc_error_codes", "rustc_error_messages", "rustc_errors",
↪  "rustc_expand", "rustc_feature", "rustc_fs_util",
↪  "rustc_graphviz", "rustc_hir", "rustc_hir_analysis",
↪  "rustc_hir_pretty", "rustc_incremental", "rustc_index",
↪  "rustc_infer", "rustc_interface", "rustc_lexer",
↪  "rustc_lint", "rustc_lint_defs", "rustc_llvm", "rustc_log",
↪  "rustc_macros", "rustc_metadata", "rustc_middle",
↪  "rustc_mir_build", "rustc_mir_dataflow",
↪  "rustc_mir_transform", "coverage_test_macros",
↪  "rustc_monomorphize", "rustc_parse", "rustc_parse_format",
↪  "rustc_passes", "rustc_plugin_impl", "rustc_privacy",
↪  "rustc_query_impl", "rustc_query_system", "rustc_resolve",
↪  "rustc_save_analysis", "rustc_serialize", "rustc_session",
↪  "rustc_smir", "rustc_span", "rustc_symbol_mangling",
↪  "rustc_target", "rustc_trait_selection", "rustc_traits",
↪  "rustc_transmute", "rustc_ty_utils", "rustc_type_ir"] stage1
test result: ok. 49 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.00s
test result: ok. 19 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.00s
test result: ok. 15 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.02s
test result: ok. 5 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.00s
test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.00s
test result: ok. 8 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.00s
test result: ok. 162 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.01s
```

```
test result: ok. 8 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.00s
test result: ok. 47 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.01s
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.00s
test result: ok. 11 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.00s
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.00s
test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.00s
test result: ok. 37 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.10s
test result: ok. 16 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.03s
test result: ok. 32 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.00s
test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.00s
test result: ok. 5 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.17s
test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.00s
test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.00s
test result: ok. 8 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.00s
test result: ok. 16 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.00s
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.00s
test result: ok. 10 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.00s
test result: ok. 22 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.01s
test result: ok. 32 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.12s
test result: ok. 200 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.01s
test result: ok. 11 passed; 0 failed; 0 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.00s
test result: ok. 3 passed; 0 failed; 2 ignored; 0 measured; 0
 ↪  filtered out; finished in 0.25s
```

```
test result: ok. 2 passed; 0 failed; 3 ignored; 0 measured; 0
↪  filtered out; finished in 0.25s
test result: ok. 10 passed; 0 failed; 4 ignored; 0 measured; 0
↪  filtered out; finished in 0.93s
test result: ok. 1 passed; 0 failed; 2 ignored; 0 measured; 0
↪  filtered out; finished in 0.18s
test result: ok. 22 passed; 0 failed; 1 ignored; 0 measured; 0
↪  filtered out; finished in 1.62s
test result: ok. 6 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.62s
test result: ok. 5 passed; 0 failed; 6 ignored; 0 measured; 0
↪  filtered out; finished in 0.37s
test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured; 0
↪  filtered out; finished in 0.19s
test result: ok. 60 passed; 0 failed; 2 ignored; 0 measured; 0
↪  filtered out; finished in 3.18s
test result: ok. 94 passed; 0 failed; 19 ignored; 0 measured; 0
↪  filtered out; finished in 3.55s
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.51s
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.05s
test result: ok. 14 passed; 0 failed; 12 ignored; 0 measured; 0
↪  filtered out; finished in 0.73s
test result: ok. 4 passed; 0 failed; 2 ignored; 0 measured; 0
↪  filtered out; finished in 0.31s
test result: ok. 3 passed; 0 failed; 14 ignored; 0 measured; 0
↪  filtered out; finished in 0.26s
test result: ok. 2 passed; 0 failed; 2 ignored; 0 measured; 0
↪  filtered out; finished in 0.23s
test result: ok. 1 passed; 0 failed; 8 ignored; 0 measured; 0
↪  filtered out; finished in 0.05s
test result: ok. 4 passed; 0 failed; 10 ignored; 0 measured; 0
↪  filtered out; finished in 0.29s
test result: ok. 2 passed; 0 failed; 3 ignored; 0 measured; 0
↪  filtered out; finished in 0.22s
test result: ok. 3 passed; 0 failed; 14 ignored; 0 measured; 0
↪  filtered out; finished in 0.25s
test result: ok. 2 passed; 0 failed; 7 ignored; 0 measured; 0
↪  filtered out; finished in 0.21s

Testing rustdoc stage1
test result: ok. 87 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.01s
```

```
test result: ok. 3 passed; 0 failed; 5 ignored; 0 measured; 0
↪  filtered out; finished in 0.64s

Testing rustdoc-json-types stage1
test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.00s
test result: ok. 7 passed; 0 failed; 2 ignored; 0 measured; 0
↪  filtered out; finished in 0.52s
test result: ok. 7 passed; 0 failed; 0 ignored; 0 measured; 0
↪  filtered out; finished in 0.08s

Check compiletest suite=run-make-fulldeps mode=run-make
test result: FAILED. 0 passed; 181 failed; 52 ignored; 0
↪  measured; 0 filtered out; finished in 0.06s
```

Bibliography

[1] Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. The Prusti Project: Formal Verification for Rust. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods*, pages 88–108, Cham, 2022. Springer International Publishing.

[2] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How Do Programmers Use Unsafe Rust? *Proc. ACM Program. Lang.*, 4(OOPSLA):1–27, November 2020.

[3] Doug Bagley, Brent Fulgham, and Isaac Gouy. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/. [Accessed 2023-07-10].

[4] Aria Beingessner. Learn Rust With Entirely Too Many Linked Lists. https://rust-unofficial.github.io/too-many-lists/. [Accessed 2023-07-10].

[5] compiletest-rs. An extraction of the compiletest utility from the Rust compiler. https://github.com/Manishearth/compiletest-rs. [Accessed 2023-07-10].

[6] LLVM Foundation. The LLVM Compiler Infrastructure. https://llvm.org/. [Accessed 2023-07-10].

[7] The Rust Foundation. The Rust Programming Language Repository. https://github.com/rust-lang/rust. [Accessed 2023-07-10].

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Boston, MA, October 1994.

[9] Rust Compiler Development Guide. rustc_driver and rustc_interface. https://rustc-dev-guide.rust-lang.org/rustc-driver.html. [Accessed 2023-07-10].

[10] Rust Compiler Development Guide. The MIR (Mid-level IR). https://rustc-dev-guide.rust-lang.org/mir/index.html. [Accessed 2023-07-10].

[11] Ralf Jung. An interpreter for Rust's mid-level intermediate representation. https://github.com/rust-lang/miri. [Accessed 2023-07-10].

[12] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked Borrows: An Aliasing Model for Rust. *Proc. ACM Program. Lang.*, 4(POPL):1–32, January 2020.

[13] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, San Francisco, CA, August 2019.

[14] Jannis Christopher Köhl. New MIR constant propagation based on dataflow analysis. https://github.com/Manishearth/compiletest-rs. [Accessed 2023-07-10].

[15] Niko Matsakis. Introducing MIR. https://blog.rust-lang.org/2016/04/19/MIR.html, April 2016. [Accessed 2023-07-10].

[16] Anders Møller and Michael Schwartzbach. *Static Program Analysis*. Department of Computer Science Aarhus University, Denmark, March 2023.

[17] The Rust Foundation. Rust By Example. https://doc.rust-lang.org/rust-by-example/index.html. [Accessed 2023-07-10].

[18] Neven Villani. Tree Borrows. https://perso.crans.org/vanille/treebor/, March 2023. [Accessed 2023-07-10].

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Ownership Typesystem based Optimisations for Rust

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Peyer	Janis

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zürich, 11.07.23

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.