ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computing Fine-grained Type Information for Rust Programs

Master Thesis

William Seddon

September 8, 2020

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

Department of Computer Science, ETH Zürich

Abstract

Understanding why variable uses are valid or invalid in Rust programs requires more information than currently exposed by the Rust compiler. Previous works in this field introduced appropriate notation but use unsuitable, unstable internal Rust compiler information as an oracle. The previous implementation does not support some common Rust cases such as function calls. This thesis describes techniques to support common Rust cases, such as loops and function calls, without relying on an external oracle. The theory successfully supports non-trivial cases such as function calls with traits, and the accompanying implementation is between $2\times$ and $3\times$ faster than the previous implementation while supporting a larger subset of Rust programs.

Acknowledgements

I am eternally grateful for the patient advice and guidance of both Federico and Vytautas, without both of whom this would not have been possible. I am also grateful to Prof. Müller for allowing me to write my master thesis in the Programming Methodology group, and to the ETH for two great exchange semesters. I would also like to thank my aunt and uncle for being such gracious hosts and putting up with me, as well as both of my parents for their loving and unwavering support.

Contents

Chapter 1

Introduction

1.1 Problem

Rust is an increasingly popular programming language designed at Mozilla Research. It attempts to improve on existing programming languages such as C and C++ by introducing new rules that give safety guarantees. These rules stem from Rust's ownership type system [2], where each variable has one owner. This ownership can be transferred, either temporarily by borrowing, or permanently by moving. The type system imposes rules, one example being that only one mutable borrow of a variable can exist at any point in time.

These rules mean that, for Rust tooling, it is no longer enough to know the types of variable to be able to deduce whether variables are allowed to be used. Tools such as Prusti, a formal verification tool for Rust developed at the ETH Zürich, need access to more fine-grained type information that takes into account the rules of the ownership type system.

The previous work in this field by Wolff in the report *Extended Place Capabilities Summaries for Rust Programs* [6] described a format for the required fine-grained type information. The format was called an Extended Place Capability Summary (EPCS), which contains a Place Capability Summary (PCS) and a Reference Capability Summary (RCS). The PCS was initially described in the Prusti paper [1] as a set of places a program is allowed to use at any one point. The previous work [6] introduces the RCS which describes the state of borrows at any one point. The report by Wolff [6] describes the methods to generate these extended place capability summaries from MIR, an intermediate form of Rust exposed by the compiler [4]. However these methods rely on information from a ”borrow oracle”, which is an oracle that explains the current state of the borrows, especially for complex situations such as loops. In the implementation of Wolff [6] the borrow oracle used is Polonius [3], the next generation Rust borrow checker. Using information from Polonius however has drawbacks, including the need to fork Polonius code to expose needed data structures. The implementation also doesn't support some common Rust cases such as function calls.

1.2 High Level Hints

The information gleaned from the borrow oracle needs to be replaced with some techniques that handle cases previously outsourced to Polonius. Common Rust cases that need to be covered include loops, enumerations, slices and function calls. To cover some of these cases approximations will need to be developed. To develop these techniques a deeper understanding of how the compiler treats borrows in certain situations is required. This deeper understanding was developed by testing the behaviour of the Rust compiler by building Rust programs especially designed to discover the limits of the compiler and the information it stores about borrows.

1.3 Contributions

This thesis describes some new techniques for describing the approximation required to support loops in the control flow graph. Function calls, where the arguments or the return value contain references, introduce constraints between these references. This thesis introduces a format to express these constraints, called function reference annotations, as well as a set of rules that generate these constraints. The format of function reference annotations has been extended to introduce support for traits with lifetimes. The Rust compiler approximates references in certain data structures such as enumerations and slices. This thesis introduces a form expressing this approximation.

Along with the theory this thesis introduces an implementation which does not use any information from Polonius or any other borrow oracle, but instead calculates the information needed itself. The implementation supports a larger subset of Rust code than the previous implementation, and in benchmarks it is between 2× and 3× faster than the previous implementation. The implementation also powers a Visual Studio Code extension that displays human readable EPCS information to end users when hovering over Rust code. As an extension the implementation was extended to support two-phase borrows, a more complicated

1.4 Structure

The first chapter, approach, introduces the theoretical elements that underpin the implementation. The second chapter describes the implementation, and the third chapter evaluates the suitability of the theory and the implementation.

Chapter 2

Approach

2.1 Fix Point Iteration

The task of traversing the MIR control flow graph to generate EPCSs for all the CFG blocks lends itself well to the ”data-flow analysis” technique. As part of the data-flow analysis each CFG block has an input EPCS and an output EPCS: the input EPCS is used as the initial EPCS which the statements in the block modify, and the block output EPCS is the EPCS of the last statement in the block. The algorithm then iterates through the CFG, following the edges between blocks, and using the output EPCS of predecessor blocks as the input EPCS of the current block. For example in figure 2.1, the input EPCS for `Block 2` will be the output EPCS of `Block 1`.

One question is what to do when a block has multiple predecessor blocks, as seen in 2.1 with `Block 4`. Through a small experiment in figure 2.2 it

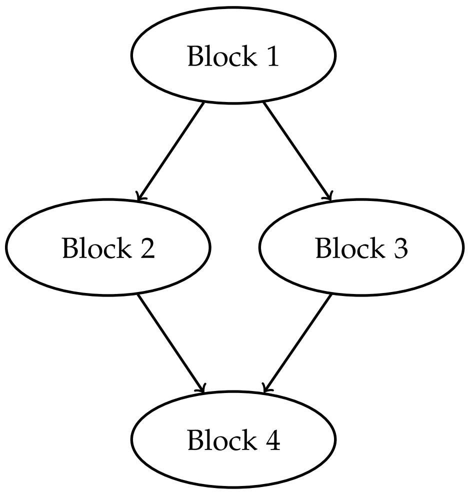


Figure 2.1: A simple control flow graph with a branch

```rust
fn takes_bool(discrim: bool) {
    let x = "Cry Havoc! And let slip the dogs of
    war".to_string();
    if discrim {
        // Do nothing
        // {x} ()
    } else {
        drop(x);
        // {} ()
    }

    drop(x); // Error as x could be moved out in the else
    branch
    // Must take into account both EPCSs. {x} () is NOT the
    correct EPCS here.
}
```

Figure 2.2: Iterating through a linked list

is clear that simply ignoring one of the predecessor blocks EPCSs is wrong, they must be merged together.

The previous work in this field introduced two simple rules for merging two incoming EPCSs:

1. Take the intersection of the allowed places.
2. Take the intersection of the references. However if the EPCSs both have a reference that points to different places then the merged reference points to the union of those places.

For example merging `{x, y, z} (z -> a)` and `{z} (z -> b)` will result in `{z} (z -> {a, b})`.

For simple branching control flows this technique works well, however when dealing with more complex cases where the control flow graph has cycles such as in figure 2.3 it is insufficient. The cyclical nature of the CFG means that calculating the output EPCS for `Loop Body` changes the input, and therefore output EPCSs of `Loop Head`, which in turn changes the input and output of `Loop Body`, which changes the input to `Loop Head` which continues ad infinitum. As part of the data-flow analysis the algorithm will continue to iterate through the CFG, following edges and calculating input and output EPCSs, until the input and output EPCSs of all blocks no longer change: i.e. we have converged to an equilibrium or fix-point. As the code in figure 2.4 shows, without some changes it is clear that there are cases where a fix-point is not reached. The previous implementation skirted around this problem as Polonius successfully handled the calculation of borrows in loops.

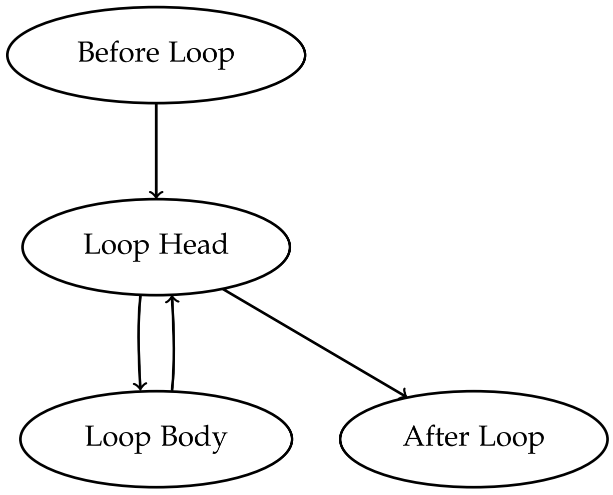


Figure 2.3: A control flow graph with a cyclical loop

Looking at the EPCSs generated in figure 2.4 it is clear that the problem lies with the fact that `cur` could point to any node in the linked list, which is potentially infinitely long. The infinite list of potential nodes isn't very useful, despite being accurate, as it is impossible to compute. It also it doesn't matter to the type system exactly which node `cur` points to, but it does matter that it points to `root` or a node further on from `root`. Rather than let `cur` point to an infinite set, by observing that `cur` will always point to something "reachable" from `root` it is possible to approximate the infinite set as `cur -> reachable(root)`.

In this context `reachable(x)` is an over-approximation, as we define it to be the set of memory locations accessible by applying a projection to `x`. The previous example has been modified in figure 2.5 to show how the EPCSs in a loop converge to a fix-point when approximating using `reachable`.

```
reachable(x) ::= x | reachable(*x) | reachable(x.field) |
↪  reachable(x as Variant.field)
```

The Rust compiler uses a very similar method to the reachable approximation internally when borrow checking.

The reachable form is entered when merging two EPCSs, if a reference points to two different things, but these targets of the references share a common prefix. In that case we approximate to something reachable from that common prefix.

It is important to note that, once the EPCS holds a reference in the reachable form, it is impossible to leave the reachable approximation and point to something concrete again.

```rust
struct LinkedListNode {
    next: Option<Box<LinkedListNode>>,
}

fn append(root: Option<Box<LinkedListNode>>) {
    let mut cur = root;
    // {cur} (cur -> root)

    // EPCS for block which decides if to enter loop
    // Before 1st iteration:  {cur} (cur -> root)
    // Before 2nd iteration:  {cur} (cur -> {root, root as Some.0.next})
    // Before 3rd iteration:  {cur} (cur -> {root, root as Some.0.next, root as Some.0.next as Some.0.next})
    // ...

    while let Some(ref mut node) = cur {
        // 1st iteration:  {node} (node -> root as Some)
        // 2nd iteration:  {node} (node -> {root as Some, root as Some.0.next as Some})
        // 3rd iteration:  {node} (node -> {root as Some, root as Some.0.next as Some, root as Some.0.next as Some.0.next as Some})
        cur = &mut node.next;
        // 1st iteration:  {cur} (cur -> root as Some.0.next)
        // 2nd iteration:  {cur} (cur -> {root as Some.0.next, root as Some.0.next as Some.0.next, root as Some.0.next as Some.0.next as Some.0.next})

        // Control flow graph points  back to the start
    }
}
```

Figure 2.4: Iterating through a linked list without approximations

```rust
struct LinkedListNode {
    next: Option<Box<LinkedListNode>>,
}

fn append(root: Option<Box<LinkedListNode>>) {
    let mut cur = root;
    // {cur} (cur -> root)

    // EPCS for block which decides if to enter loop
    // Before 1st iteration:  {cur} (cur -> root)
    // After 1st iteration:  {cur} (cur -> root as
    ↪   Some.0.next})
    // Merged/2nd iteration:  {cur} (cur -> reachable(root))
    // ...

    while let Some(ref mut node) = cur {
        // 1st iteration:  {node} (node -> root as Some)
        // 2nd iteration:  {node} (node -> reachable(root))
        cur = &mut node.next;
        // 1st iteration:  {cur} (cur -> root as Some.0.next)
        // 2nd iteration:  {cur} (cur -> reachable(root))

        // Here our control flow loops back to the start
    }
}
```

Figure 2.5: Iterating through a linked list with the reachable approximation

```rust
// {a} (a -> reachable(root))
b = &mut a.next;
// {b} (b -> reachable(root))
```

Figure 2.6: Excerpt from a Rust program showing how borrowing a field of a struct behind a reference in the `reachable` approximation also results in a borrow with a `reachable` approximation

```
PCS 1: x -> root
PCS 2: x -> root as Some.0
Merged EPCS: x -> reachable(root)
```

Figure 2.7: Example of two EPCSs being merged into one EPCS with a `reachable` approximation

Left	Right	Merged
`x`	`x.y`	`reachable(x)`
`x.y`	`reachable(x)`	`reachable(x)`
`x`	`y`	`{x, y}`
`x`	`reachable(x)`	`reachable(x)`
`reachable(x)`	`reachable(x)`	`reachable(x)`
`x`	`reachable(y)`	`{x, reachable(y)}`
`x.z`	`reachable(y)`	`{x.z, reachable(y)}`
`reachable(x)`	`reachable(y)`	`{reachable(x), reachable(y)}`
`{x, y}`	`{y, z}`	`{x, y, z}`

Figure 2.8: Informal table of rules for merging EPCSs with `reachable`

2.2 Refs Notation

In Rust programs that use certain data structures it becomes unclear which references in the structures point to which places. One such case is with data structures, such as enums, that do not support split borrows. For a data structure to support split borrows the Rust compiler has to be intelligent enough to see that it is possible to simultaneously borrow disjoint fields. Figure 2.2 shows an example of the compiler supporting split borrows on a `struct`. The notable data structures which do not support split borrows are

```rust
struct S<'a, 'b, 'c> {
    one: String,
    two: String,
    three: String,
}

fn main() {
    let mut t = S{
        one: "tomorrow, and tomorrow, and tomorrow".to_string(),
        two: "creeps in this petty pace".to_string(),
        three: "from day to day".to_string(),
    };

    let x = &mut t.one;
    let y = &mut t.two;
    // This is allowed!
}
```

Figure 2.9: Rust program showing how the compiler supports split borrowing of distinct struct fields

enumerations, slices and arrays. In the case of enumerations this is partly because at a Rust level individual enumeration fields cannot be accessed, so pattern matching must be used to extract data. There is however an important distinction to be made between the levels of Rust and MIR: while in Rust enumerations can only be accessed by pattern matching, at the MIR level enumerations are accessed using fields in a similar way to tuples or anonymous structures in the form `variable as Variant.field`. Despite being accessed in a similar way to structures at the MIR level there is no support for split borrows on enumerations.

This lack of split borrowing affects the EPCS in a rather subtle way. Figure 2.10 shows how the intelligent borrow checker knows the use of `t.two` is still valid because only a use that includes the field `t.one` is invalid. However, as figure 2.2 shows, the compiler is not able to reason about enumerations in the same way. The compiler takes an approximation and forgets exactly what each enumeration field points to. The fact that the code snippet in figure 2.12 passes the borrow checker shows that the approximation is lifetime dependent; the compiler groups the borrows together by lifetime. The lifetimes for a data structure are defined at the same time a data structure is defined, for example `struct T<'a, 'b>` defines a structure `T` with two lifetimes `'a` and `'b` which have no relationship.

It is not just the case that the compiler groups this approximation by lifetime, but it also takes into account which lifetimes outlive others. A lifetime can be defined to outlive another one: for example `struct T<'a, 'b: 'a>` states that lifetime `'b` outlives `'a`, which means all references in `T` with lifetime `'b` will "live" or be usable at least as long as all references with lifetime `'a`. The relation, outlives, is transitive: for

```
struct T<'a, 'b: 'a, 'c: 'b>
```

`'c` outlives `'a` because `'c: 'b` and `'b: 'a`.

$$\forall a.b.c.\ \texttt{'c} : \texttt{'b} \land \texttt{'b} : \texttt{'a} \implies \texttt{'c} : \texttt{'a}$$

To simplify matters a lifetime is also defined to outlive itself. The Rust program in figure 2.17 shows that the compiler takes into account the relationship between lifetimes when approximating non-split borrow data structures.

A simple option to represent this situation would be to approximate each field or index of the place individually, as shown in figure 2.13. This approach is however incorrect as it implies that it is possible to escape the approximate form by reassigning an individual field or element. Figure 2.13 provides a counter example that would compile if this approach were correct, but it does not compile.

```rust
struct T<'a> {
    one: &'a String,
    two: &'a String,
    three: &'a String,
}

fn main() {
    let str_1 = "tomorrow, and tomorrow, and
↪   tomorrow".to_string();
    let str_2 = "creeps in this petty pace".to_string();
    let str_3 = "from day to day".to_string();

    let mut t = T{one: &str_1, two: &str_2, three: &str_3};
    // {t, str_1, str_2, str_3} (t.one -> str_1, t.two -> str_2,
↪   t.three -> str_3)

    drop(str_1);
    // {t.two, t.three, str_2, str_3} (t.two -> str_2, t.three
↪   -> str_3)

    drop(t.two); // Works!

    // drop(t.one) would fail because str_1 has been moved out
}
```

Figure 2.10: Rust program showing how a structure that supports split-borrows allows access to non-moved out fields while other fields have been moved out

Therefore the notation to describe the approximation should be more general to show that it is impossible to escape the approximate form. In light of these experiments a new form of EPCS statement is required. Instead of reasoning about each field or index individually it makes more sense to reason about references on a lifetime level, being able to express ”all references in a structure with a lifetime that outlives ’a”.

To this end the notation `refs(x, ’a)` is introduced, which is a notation for the set of all references in `x` (including `x` itself if `x` is a reference) which have a lifetime of `’a` or outlive lifetime `’a`. To give some intuition as to how this new EPCS approximation works figure 2.14 gives an annotated example of the state of the references in the EPCS using this `refs` approximation.

If in a type definition lifetime `’b` is defined to outlive `’a` then `refs(x, ’a)` is a superset of `refs(x, ’b)` for any variable `x` of that type.

$$\forall a.b.x.\ \texttt{'b} : \texttt{'a} \implies \texttt{refs}(x, \texttt{'a}) \supset \texttt{refs}(x, \texttt{'b})$$

```rust
enum E<'a> {
    Inner(&'a String, &'a String, &String),
}

fn main() {
    let str_1 = "tomorrow, and tomorrow, and tomorrow".to_string();
    let str_2 = "creeps in this petty pace".to_string();
    let str_3 = "from day to day".to_string();

    let mut e = E::Inner(&str_1, &str_2, &str_3);

    if let E::Inner(a, b, c) = e {
        drop(str_1);
        drop(b); // Error! Value behind borrow moved out
        // The compiler has forgotten that e.0 -> str_1 and e.1 -> str_2
    }
}
```

Figure 2.11: Rust program, when compared to figure 2.2 shows how an enumeration that does not support split-borrows doesn't allow access to non-moved out fields when the borrows of fields have been moved out

```rust
enum E<'a, 'b> {
    Inner(&'a String, &'a String, &'b String),
}

fn main() {
    let str_1 = "tomorrow, and tomorrow, and tomorrow".to_string();
    let str_2 = "creeps in this petty pace".to_string();
    let str_3 = "to the last syllable of recorded time".to_string();

    let mut e = E::Inner(&str_1, &str_2, &str_3);

    if let E::Inner(a, b, c) = e {
        drop(str_1);
        drop(c); // Works!
        drop(b); // Error! Value behind borrow moved out
    }
}
```

Figure 2.12: Rust program showing how which fields on an enumeration can be used after moving out a borrow depends on the lifetime of the field

```rust
fn main() {
    let str_1 = "tomorrow, and tomorrow, and tomorrow".to_string();
    let str_2 = "creeps in this petty pace".to_string();
    let str_3 = "to the last syllable of recorded time".to_string();

    let mut x = [&str_1, &str_2, &str_3];
    // x[0] -> {str_1, str_2, str_3}, x[1] -> {str_1, str_2, str_3}, x[2] -> {str_1, str_2, str_3}

    x[0] = &str_3;
    // x[0] -> str_3, x[1] -> {str_1, str_2, str_3}, x[2] -> {str_1, str_2, str_3}

    drop(str_1);
    // If the notation above is correct then this statement should be fine
    // as we know x[0] -> str_3
    drop(x[0]); // Error!
}
```

Figure 2.13: Rust code showing how reassigning does not allow escape from the approximation

```rust
fn main() {
    let str_1 = "tomorrow, and tomorrow, and tomorrow".to_string();
    let str_2 = "creeps in this petty pace".to_string();
    let str_3 = "to the last syllable of recorded time".to_string();

    let mut x: [&String; 3] = [&str_1, &str_2, &str_3];
    // There is an anonymous lifetime associated with x
    // Really the type looks like [&'1 String; 3]

    // refs(x, '1) -> {str_1, str_2, str_3}

    drop(str_1);
    drop(x[1]); // Error! Use of moved out value
}
```

Figure 2.14: Annotated Rust program showcasing the `refs` approximation for slices

```rust
enum E<'a> {
    Inner(&'a String, &'a String),
}

fn main() {
    let str_1 = "tomorrow, and tomorrow, and
↪   tomorrow".to_string();
    // {str_1} ()
    let str_2 = "creeps in this petty pace".to_string();
    // {str_2, str_1} ()

    let e = E::Inner(&str_1, &str_2);
    // {e, str_1, str_2} (refs(e, 'a) -> {str_1, str_2})

    if let E::Inner(a, b) = e {
        // {a, b, str_1, str_2} (refs(a, 'a) -> {str_1, str_2},
↪   refs(b, 'a) -> {str_1, str_2})
        // this is equiv to (a -> {str_1, str_2}, b -> {str_1,
↪   str_2})
        // as a: &'a String, and b: &'a String
        drop(str_1);
        // {str_2} ()
        drop(b); // Error! Value behind borrow moved out
    }
}
```

Figure 2.15: Annotated Rust program showing the approximation of references in an enumeration with only one lifetime

```rust
enum E<'a, 'b> {
    Inner(&'a String, &'a String, &'b String),
}

fn main() {
    let str_1 = "tomorrow, and tomorrow, and
↪   tomorrow".to_string();
    let str_2 = "creeps in this petty pace".to_string();
    let str_3 = "to the last syllable of recorded
↪   time".to_string();

    // {str_3, str_2, str_1} ()

    let e = E::Inner(&str_1, &str_2, &str_3);
    // {e, str_1, str_2, str_3} (refs(e, 'a) -> {str_1, str_2},
↪   refs(e, 'b) -> str_3)

    if let E::Inner(a, b, c) = e {
        // {a, b, c, str_1, str_2, str_3} (refs(a, 'a) ->
↪   {str_1, str_2}, refs(b, 'a) -> {str_1, str_2}, refs(c, 'b)
↪   -> str_3)
        // this is equiv to (a -> {str_1, str_2}, b -> {str_1,
↪   str_2}, c -> str_3)
        // as a: &'a String, and b: &'a String and c: &'b
↪   String

        // {a, b, c, str_1, str_2, str_3} (a -> {str_1, str_2},
↪   b -> {str_1, str_2}, c -> str_3)
        drop(str_1);
        // {c, str_2, str_3} (c -> str_3)
        drop(c); // Works!
        drop(b); // Error! Value behind borrow moved out
    }
}
```

Figure 2.16: Annotated Rust program showing the approximation of references in an enumeration with two unrelated lifetimes

```rust
enum E<'a: 'b, 'b> {
    Inner(&'a String, &'a String, &'b String),
}

fn main() {
    let str_1 = "tomorrow, and tomorrow, and tomorrow".to_string();
    let str_2 = "creeps in this petty pace".to_string();
    let str_3 = "to the last syllable of recorded time".to_string();

    // {str_3, str_2, str_1} ()

    let e = E::Inner(&str_1, &str_2, &str_3);
    // {e, str_1, str_2, str_3} (refs(e, 'a) -> {str_1, str_2}, refs(e, 'b) -> {str_1, str_2, str_3})

    if let E::Inner(a, b, c) = e {
        // refs(a, 'a) -> {str_1, str_2}, refs(b, 'a) -> {str_1, str_2}, refs(c, 'b) -> {str_1, str_2, str_3}
        // this is equiv to (a -> {str_1, str_2}, b -> {str_1, str_2}, c -> {str_1, str_2, str_3})
        // as a: &'a String, and b: &'a String and c: &'b String

        // {a, b, c, str_1, str_2, str_3} (a -> {str_1, str_2}, b -> {str_1, str_2}, c -> {str_1, str_2, str_3})
        drop(str_1);
        // {str_2, str_3} ()
        drop(c); // Error! Value behind borrow moved out
    }
}
```

Figure 2.17: Annotated Rust program showing how the lifetime relationship `'a: 'b` means that if any borrows with lifetime `'a` are moved out then borrows with lifetime `'b` are invalid

2.3 Function Reference Annotations

2.3.1 Background

The reason for the introduction of the `refs(x, 'a)` approximation is that there are cases where it is no longer possible to be precise about where each borrow points to. Apart from the already stated case of non split borrow data structures the main source of reference uncertainty is function calls. Consider the situation in figure 2.18; what exactly `left.inner` and `right.inner` point to after the call to `swap` depends on the implementation of the function `swap`. It is however infeasible for a tool to check the implementation of every function called, not only would it be incredibly inefficient but if linked libraries are used the compiler may not have the source code or even the machine code of the implementation available at compile time.

Rather than process every function call body the only remaining possibility is to infer the possible effects of the function through analysis of its function signature alone. This is exactly what the Rust compiler does, it uses the function signature to over-approximate what the function could do to the references in its arguments and then enforces these constraints. The example code in figure 2.19 shows that even if the `swap` function is a no-op the compiler still enforces the constraints that it has derived from the function signature. This proves that it is the function signature, not the function body, that the Rust compiler uses when borrow checking function calls.

Rust function signatures, like other Rust statements that define lifetimes such as `struct` definitions, can define lifetimes which are used in the signature. They can also define constraints between the lifetimes, for example in signature

```rust
fn foo<'a: 'b, 'b>(left: &'a i32, right: &'a i32) -> &'b i32
```

the lifetime `'a` outlives the lifetime `'b`, which in this example means the return value could point to either `*left` or `*right`.

2.3.2 Framework

Function signatures in Rust can range from the very simple, as seen in previous examples, to incredibly complex signatures with traits that require a logic solver, such as Chalk, to resolve. Functions create constraints and relationships between the references in their arguments and their return value, and as seen before these relationships must be calculated from the function signature. The principle behind function reference annotations (FRA) is to specify a format for expressing these relationships and a set of rules to calculate the relationships from a function signature.

```rust
struct T<'a> {
    inner: &'a String,
}

fn swap<'a>(left: &mut T<'a>, right: &mut T<'a>) {
    left.inner = right.inner;
}

fn main() {
    let str_1 = "The noble Brutus".to_string();
    let str_2 = "Hath told you Caesar was
↪   ambitious".to_string();

    let mut left = T{inner: &str_1};
    // left.inner -> str_1
    let mut right = T{inner: &str_2};
    // right.inner -> str_2

    swap(&mut left, &mut right);
    // What left and right point to depends on the
    // implementation of swap. It could also validly be
    // fn swap<'a>(left: &mut T<'a>, right: &mut T<'a>) {}

    drop(str_2);
    drop(left.inner); // Error! Borrowed value moved out
}
```

Figure 2.18: Rust program with annotations showing how calling a simple function affects the compilers knowledge of references

Informally a function can have between none and many individual annotations. An annotation takes the form of:

$$\texttt{refs\_after}(x,'a) := \texttt{refs\_before}(y,'b)\ [\cup\ \texttt{refs\_before}(z,'c)]*$$

Similar to `refs(x, ’a)`, `refs_before(x, ’a)` represents the set of references in `x` that outlive `’a`, `refs_before` however reflects the state of these references just before the function call. `refs_after(x, ’a)`, again similar to `refs(x, ’a)` represents the set of references in `x` with a lifetime that outlives `’a`, the `refs_after` statement represents the state of these references immediately after the function call has returned. The annotation shows how the references in variables after the function call could come from references before the function call. The Rust compiler occasionally references the ”flow” of references in borrow checker error messages, and it may be helpful to

```rust
struct T<'a> {
    inner: &'a String,
}

fn swap<'a>(left: &mut T<'a>, right: &mut T<'a>) {}

fn main() {
    let str_1 = "The noble Brutus".to_string();
    let str_2 = "Hath told you Caesar was
↪   ambitious".to_string();

    let mut left = T{inner: &str_1};
    // left.inner -> str_1
    let mut right = T{inner: &str_2};
    // right.inner -> str_2

    swap(&mut left, &mut right);
    // A human knows: left.inner -> str_1, right.inner -> str_2
    // Tooling knows: left.inner -> {str_1, str_2}, right.inner
↪   -> {str_1, str_2}

    drop(str_1);
    drop(right.inner); // Error! Borrowed value moved out
}
```

Figure 2.19: Rust program with annotations showing how even a no-op function imposes constraints on the arguments, proving it is the function signature that is analysed, not the function body

```rust
// refs_after(return value, 'a) := refs_before(input, 'a)
fn identity<'a>(input: &'a i32) -> &'a i32
```

Figure 2.20: Function reference annotation for a simple identity function

```rust
// refs_after(return value, 'a) := refs_before(left, 'a) ∪
↪   refs_before(right, 'a)
fn choose(left: &'a i32, right: &'a i32) -> &'a i32
```

Figure 2.21: Function reference annotation for a function which returns one of two arguments

think that references can flow from the references on the right hand side of an annotation to the left hand side due to the function.

```
// refs_after(return value, 'a) := refs_before(left, 'a)
fn choose_left<'a, 'b>(left: &'a i32, right: &'b i32) -> &'a i32
```

Figure 2.22: Function reference annotation for a function which, due to the defined lifetimes, can only return the first argument

2.3.3 Rules

To generate the annotations a number of rules are defined. The rules given here are not exhaustive, there are complex cases that were deemed out of scope.

Each rule will generate one or many annotations. If a rule generates an annotation which shares a left hand side (`refs_after(x, 'a)`) with another generated annotation the two are simply merged by taking a union of the two right hand sides. In practice merging

```
refs_after(x, 'a) := refs_before(y, 'b)
```

and

```
refs_after(x, 'a) := refs_before(z, 'c) ∪ refs_before(zy, 'd)
```

makes

```
refs_after(x, 'a) := refs_before(y, 'b) ∪ refs_before(z, 'c) ∪ refs_before(zy, 'd)
```

When encountering `refs_before` statements with related lifetimes, because of the shared heritage from `refs`,

```
'a: 'b ⟹ refs_before(x, 'b) ⊂ refs_before(x, 'a)
```

the `refs_before` with the longer lifetime can be left out.

The Rust compiler allows lifetimes to be left out of the function signatures in cases where it can infer the correct lifetimes. Rules should be generated on function signatures with the lifetimes elided by lifetime elision explicitly in the function signature.

Relations between between references in the function arguments or return values do not depend on the types of these references, only the lifetimes. Take the example in figure 2.23, the reason why the Rust compiler supports cases such as `imm_to_mut` is that the function may be implemented with unsafe code, so these relationships could still be created by the function.

Rule 1: Arguments with mutable references

It is possible for a function to change the references in the arguments passed to it, through the use of mutable borrows. Any reference that is "after" or

```rust
// Impossible to do in completely safe Rust
fn imm_to_mut<'a>(one: &'a String) -> Option<&'a mut i32> { None
↪ }

fn main() {
    let x = "Ill met by moonlight, proud Titania".to_string();
    let y = imm_to_mut(&a);
    drop(x);
    drop(y); // Error! Use of borrow after moving out
}
```

Figure 2.23: Rust counterexample showing how only the lifetimes of references are important to create function reference annotations

```rust
// refs_after(what, 'a) := refs_before(what, 'a)
fn mutate<'a>(what: &mut T<'a>)

// refs_after(left, 'a) := refs_before(left, 'a) |∪|
↪ refs_before(right, 'a)
fn swap<'a>(left: &mut &'a T, right: &'a T)

// refs_after(left, 'a) := refs_before(left, 'a) |∪|
↪ refs_before(right, 'a)
fn swap<'a>(left: &mut T<'a>, right: &T<'a>)

// refs_after(one, 'b) := refs_before(one, 'b) U
↪ refs_before(two, 'd)
fn swap<'a, 'b, 'c, 'd: 'c>(one: &'a mut S<'b>, two &'c S<'d>)
```

Figure 2.24: Examples of some of the cases handled by Rule 1

behind a mutable borrow in the type of an argument could be changed by the function.

Figure 2.24 shows some examples of Rust function signatures and the appropriate function annotations to give some intuition as to the cases covered by rule 1.

Figure 2.25 gives some Python-esque pseudocode that generates rule 1 function annotations.

Rule 2: Return arguments

The return value of a function may be one or more references. In this case a reference in the return value may point to the same place a reference in an argument points to. This could be the case if the lifetime of an argument ref-

```
# Returns a list of lifetimes which occur after the first
↪  mutable reference in an arg
# lifetimes_after_mutable(&'a &'b mut &'c &'d T<'e>) = ['c, 'd,
↪  'e]
def lifetimes_after_mutable(arg):
    def recurs(type, past_mutable):
        if type is MutRef and not past_mutable:
            return recurs(type.inner, True)
        if type is Struct and not past_mutable:
            return []
        if type is Struct:
            return type.lifetimes

        if past_mutable:
            return type.lifetimes + type.inner_types.map(inner
            ↪  recurs(inner, past_mutable))
        else:
            return type.inner_types.map(inner recurs(inner,
            ↪  past_mutable))

    recurs(arg.type, False)

# Returns list of all lifetimes in a argument
# lifetimes(&'a &'b T<'c, 'd>) = ['a, 'b, 'c, 'd]
def lifetimes(arg):
    if arg is Struct:
        return arg.lifetimes
    return arg.lifetimes + lifetimes(arg.inner)

def generate_rule_1_fra():
    for arg in all_args:
        for other_arg in all_args:
            for arg_mut_lifetime in
            ↪  lifetimes_after_mutable(arg):
                for other_lifetime in lifetimes(other_arg):
                    if outlives(other_lifetime,
                    ↪  arg_mut_lifetimes):
                        refs_after(arg, arg_mut_lifetime) ∪ =
                        ↪  refs_before(other, other_lifetime)
```

Figure 2.25: Rule 1 Python-esque pseudocode

```python
# Returns list of all lifetimes in a argument
# lifetimes(&'a &'b T<'c, 'd>) = ['a, 'b, 'c, 'd]
def lifetimes(arg):
    if arg is Struct:
        return arg.lifetimes
    return arg.lifetimes + lifetimes(arg.inner)

def generate_rule_2_fra():
    return_lifetimes = lifetimes(return_type)
    for arg in all_args:
        for arg_lifetime in lifetimes(arg):
                for return_lifetime in return_lifetimes:
                    if outlives(arg_lifetime, return_lifetime):
                        refs_after(return_value,
                        ↪  return_lifetime) ∪ = refs(arg,
                        ↪  arg_lifetime)
```

Figure 2.26: Rule 2 Python-esque pseudocode

erence outlives the lifetime of the return value reference. A simple example is

```
fn choose<'a>(l: &'a i32, r: &'a i32) -> &'a i32
```

where the return value could point to either `*left` or `*right`.

Rule 2 will generate the annotation

```
refs_after(return value, 'a) := refs_before(l, 'a) U
↪  refs_before(r, 'a)
```

for the `choose` function.

Rule 3: Returning references that could flow from arguments with generic types

When functions accept arguments with generic types, for example `fn foo<'a, T: 'a>(left: &T) -> &'a String`, the generic type may be defined to outlive a lifetime, `T: 'a`. This means that all the references in `T` will outlive the lifetime `'a`, and for the function `foo` the return value may point to something that is reachable from whatever the generic type `T` becomes monomorphised to. It does not matter that, in this example, there is no way for the function `foo` to extract a `&'a String` from a generic type which implements no traits, the relationship between the return value and the argument is still made. .

Due to `T: 'a` meaning that *all* lifetimes in generic type `T` outlive `'a` a way to express "all references in a generic type T" is needed. To this end a special lifetime signifier is used: `'* in T`. Using `'* in T` means that "all refer-

```
# Returns list of all lifetimes in a argument
# lifetimes(&'a &'b T<'c, 'd>) = ['a, 'b, 'c, 'd]
def lifetimes(arg):
    if arg is Struct:
        return arg.lifetimes
    return arg.lifetimes + lifetimes(arg.inner)

# Returns the generic types in an argument
# generic_types(&'a &'b (&'c T, &'d U)) = [T, U]
def generic_types(arg):
    if arg is GenericType:
        return arg.generic_type()
    if arg.has_inner_types():
        return arg.inner_types.map(type generic_types(type))
    return []

def generate_rule_3_fra():
    return_lifetimes = lifetimes(return_type)
    for arg in all_args:
        for arg_generic_type in generic_types(arg):
                for return_lifetime in return_lifetimes:
                    if outlives(arg_generic_type,
                     ↪ return_lifetime):
                        refs_after(return_value,
                         ↪ return_lifetime) ∪ = refs(arg, '*
                         ↪ in arg_generic_type)
```

Figure 2.27: Rule 3 Python-esque pseudocode

ences in generic type T" can be expressed as `refs(x, '* in T)`, although so far no need has been found for using this in any expression other than `refs_before(x, '* in T)`.

```
// refs_after(return_value, 'b) := refs_before(left, '* in U) U
↪  refs_before(right, '* in U)
fn foo<'a, 'b, U: 'b>(left: &'a U, right: &'a U) -> &'b i32
```

Rule 4: Generic arguments behind mutable references

In a similar vein to how the principle behind rule 2 was extended to support returning references which could flow from arguments with generic types rule 1 is now built upon. Rule 4 aims to support arguments with mutable borrows where references could flow from arguments with generic types.

Figure 2.31 gives an example of a function rule 4 supports.

```
// refs_after(right, 'b) := refs_before(left, '* in U)
fn foo<'a, 'b, U: 'b>(left: &'a U, right: &'a mut T<'b>)
```

Figure 2.28: Rust program showing that the compiler imposes constraints on the arguments of functions when using traits

Figure 2.29 provides Python-esque pseudocode for rule 4.

Rule 5: Static catch-all

It is possible to have functions return references which did not originate from any function arguments. There are two main ways that this occurs, the first being the returning of a reference with a special lifetime `'static`, as shown in figure 2.30. The `'static` lifetime is a lifetime that lasts for the entire length of the program, and therefore outlives all other lifetimes. The other way is through the use of unsafe code, a prime example being the `Box::leak` function which turns a `Box<T>` struct into `&'static mut T`.

To this end a new type of catch-all syntax is introduced: `black_box(static)`. This represents the fact that after a function call a reference could point to something leaked from a heap-allocated pointer, called a `Box` in Rust, or a static constant.

2.3.4 Applying function reference annotations to the EPCS

To go from

`refs\_after(x, 'a) := refs\_before(x, 'a) ∪ refs\_before(y, 'b)`

to a concrete reference capability summary (RCS) borrow statement such as `refs(x, 'a) -> a, b, c` there are one or two intermediate steps. `refs_before(x, 'a)` gets transformed to `reachable(*refs(x, 'a))`, when in turn the `*refs(x, 'a)` gets expanded using the information in the EPCS immediately before the function call, if the expansion of `reachable()` is "small" then that could be expanded too, this is however a personal preference. The `:=` gets transformed into a `->`. The `refs_after(x, 'a)` gets transformed into `refs(x, 'a)`. This can now be inserted into the RCS.

For function reference annotations that are in a generic form, such as `refs_before(x, '* in T)`, require information about the concrete type of `x` at the call site. With this concrete type information `refs_before(x, '* in T)` can be monomorphised to the union of multiple `refs_before(x, 'a)` statements for each lifetime in the concrete type of `x`.

```
# Returns a list of lifetimes which occur after the first mutable reference in an arg
# lifetimes_after_mutable(&'a &'b mut &'c &'d T<'e>) = ['c, 'd, 'e]
def lifetimes_after_mutable(arg):
    def recurs(type, past_mutable):
        if type is MutRef and not past_mutable:
            return recurs(type.inner, True)
        if type is Struct and not past_mutable:
            return []
        if type is Struct:
            return type.lifetimes

        if past_mutable:
            return type.lifetimes + type.inner_types.map(inner recurs(inner, past_mutable))
        else:
            return type.inner_types.map(inner recurs(inner, past_mutable))

    recurs(arg.type, False)

def generic_types(arg):
    if arg is GenericType:
        return arg.generic_type()
    if arg.has_inner_types():
        return arg.inner_types.map(type generic_types(type))
    return []
def generate_rule_4_fra():
    for arg in all_args:
        for other_arg in all_args:
            for arg_mut_lifetime in lifetimes_after_mutable(arg):
                for other_generic_type in generic_types(other_arg):
                    if outlives(other_generic_type, arg_mut_lifetimes):
                        refs_after(arg, arg_mut_lifetime) ∪ = refs_before(other, '* in other_generic_type)
```

Figure 2.29: Rule 4 Python-esque pseudocode

```rust
const s: &'static str = "And swim to yonder point?";
// refs_after(return, 'a) := black_box(static)
fn return_static<'a>() -> &'a str {
    s
}

// refs_after(return, 'a) := black_box(static)
fn return_leak<'a>() -> &'a String {
    Box::leak(Box::new("Help me Cassius or I sink".to_string()))
}
```

Figure 2.30: Two Rust functions showing how in certain cases references can be created from nothing

2.3.5 Extension of function reference annotations with traits

Rules 1 and 2 deal with most simple function cases where the argument types are fully known at compile time, however once functions start accepting generic types some more logic is required. The Rust language allows the common functionality of types to be abstracted into well defined interfaces called traits. Traits can be implemented by types, being somewhat analogous to interfaces in other programming languages such as Java or C#.

While being defined traits may also have lifetime arguments which can be used in the functions of that trait. When a trait is being implemented by a type a relationship between the trait lifetime and a lifetime in the type may be defined. When functions are defined that take arguments with generic types that implement a certain set of traits the relationship between the trait lifetimes will also create constraints on the references in the concrete types. The Rust program in figure 2.31 is a minimal example in a similar vein to previous examples, but the arguments are references to generic types which implement a trait. The comments explain the extra challenge in dealing with generic arguments.

When looking at a function signature that contains generic arguments which implement traits, the constraints imposed by the signature are in a generic form. That is to say, when analysing the signature in isolation, without knowledge of the concrete types at the call sites, the function annotation references the traits. Figure 2.32 and figure 2.33 show two function signatures with generic arguments, and give the appropriate function reference annotation in a generic form.

To apply these generic function annotations to a EPCS they must first be monomorphised. The generic arguments must be replaced with the concrete types of the arguments at the call site. The links between the concrete type at the call site and the generic arguments are the trait implementation

```rust
struct S<'a> {
    inner: &'a String,
}

trait TestTrait<'inner> {}

// This is the link between the trait lifetime and the struct lifetime
impl<'inner> TestTrait<'inner> for S<'inner> {}

// The constraints of this function signature involve generics as it references traits.
// To apply the constraints created by this function signature to the EPCS first it needs to be monomorphised by replacing the generic traits with
// the types and the lifetimes of the types at the call sites.
fn foo<'a, T: TestTrait<'a>, U: TestTrait<'a>>(left: &'a mut T, right: &'a U) {}

fn main() {
    let s1 = "All hail, Macbeth".to_string();
    let s2 = "hail to thee, thane of Cawdor".to_string();

    let mut x = S{ inner: &s1 };
    let y = S{ inner: &s2 };

    foo(&mut x, &y);
    // To apply constraints here we need to know the link between S<'a> (the type of x) and TestTrait<'inner> (the type in the function signature)
    // It is the implementation of TestTrait<'inner> for S<'inner> that shows that 'inner == 'a
    // So any constraint imposed by the function signature on 'inner (for TestTrait<'inner>) will actually impose constraints on 'a (for S<'a>)

    drop(s2);
    drop(x); // Error!
}
```

Figure 2.31: Rust program showing that the compiler imposes constraints on the arguments of functions when using traits

```
// refs_after(left as TestTrait<'b>, 'b) = refs_before(left as
↪  TestTrait<'b>, 'b) U refs_before(right as TestTrait<'b>,
↪  'b)
fn foo<'a, 'b, T: TestTrait<'b>, U: TestTrait<'b>>(left: &'a mut
↪  T, right: &'b U) {}
```

Figure 2.32: Rust function signature with the function annotation in a generic form

```
// refs_after(left as TestTrait<'c>, 'c) = refs_before(left as
↪  TestTrait<'c>, 'c) U refs_before(right as TestTrait2<'d>,
↪  'd)
fn foo<'a, 'b, 'c, 'd: 'c, T: TestTrait<'c>, U:
↪  TestTrait2<'d>>(left: &'a mut T, right: &'b U) {}
```

Figure 2.33: A more complex Rust function signature with the function annotation in a generic form

definitions. In order to express these links the trait implementation definitions are annotated to express how function reference annotations can be monomorphised using the lifetime relationships in the implementation definition.

Trait implementation annotations are directional; depending on the defined lifetime relationships there are three cases:

1. Trait functions can take a reference as an argument and put it into the data structure. The implementation

   ```
   impl<'struct, 'trait: 'struct> TestTrait<'trait> for
   ↪  S<'struct>
   ```

   can have functions that take references with `'trait` and can modify `S` to make a reference in `S` with lifetime `'struct` point to the same place as the passed reference. In this case, because it is possible to modify references in `S` using `TestTrait` `refs_after(S<'struct> as TestTrait<'trait>, 'trait)` can be monomorphised as `refs_after(S<'struct>, 'struct)`.

2. Trait functions can return a reference which comes from the data structure. The implementation

   ```
   impl<'struct: 'trait, 'trait> TestTrait<'trait> for
   ↪  S<'struct>
   ```

   can have functions that return a reference in `S` with lifetime `'struct`, the lifetime of the return reference is `'trait`. In this case this insight means that `refs_before(S<'struct> as TestTrait<'trait>, 'trait)` can be monomorphised as `refs_before(S<'struct>, 'struct)`.

   This is valid because `'struct` outlives `'trait`.

```
// refs_after(S<'a> as TestTrait<'inner>, 'inner) ->
↪  refs_after(S<'struct>, 'struct)
impl<'struct, 'inner: 'struct> TestTrait<'inner> for S<'struct>
↪  {}

// refs_before(S<'a> as TestTrait<'inner>, 'inner) ->
↪  refs_before(S<'struct>, 'struct)
impl<'struct: 'inner, 'inner> TestTrait<'inner> for S<'struct>
↪  {}

// refs_after(S<'struct> as TestTrait<'inner>, 'inner) ->
↪  refs_after(S<'struct>, 'struct)
// refs_before(S<'struct> as TestTrait<'inner>, 'inner) ->
↪  refs_before(S<'struct>, 'struct)
impl<'struct: 'inner, 'inner> TestTrait<'inner> for S<'struct>
↪  {}
```

Figure 2.34: Rust trait implementation with annotation stating that this implementation can be used to monomorphise a generic function annotation

3. The third case combines both of the above. An example of a trait implementation definition where references can flow both from trait functions to the struct and from the struct to trait functions is

   ```
   impl<'struct: 'trait, 'trait: 'struct> TestTrait<'trait>
   ↪  for S<'struct>
   ```

The form of implementation annotation is directional, showing how the generic `refs` on the left which is defined in terms of traits can be converted into a `refs` defined just in terms of concrete types: `refs_after(S<'a> as TestTrait<'inner>, 'inner) -> refs_after(S<'a>, 'a)`.

To monomorphise a function annotation that contains traits the process is to use the directional annotations on the trait implementation blocks to replace the generic `refs` relationships with their concrete equivalents. There is one edge case: it is possible for the function to not change any references, hence the relationship `refs_after(left as TestTrait<'trait>, 'trait) := refs_before(left as TestTrait<'trait>, 'trait) ....` This is not entirely accurate: `refs_before(left as TestTrait<'c>, 'c)` implies that the references come from the trait `TestTrait`, but this is not the case, the references in `left` are simply unchanged. This inaccuracy becomes a problem if a trait only allows references to flow from a trait function to the data structure, e.g. `impl<'struct, 'inner: 'struct> TestTrait<'inner> for S<'struct>` has the annotation `refs_after(S<'a> as TestTrait<'inner>, 'inner) -> refs_after(S<'struct>, 'struct)` which means there is no way to monomorphise `refs_before(left as TestTrait<'trait>, 'trait)`.

To overcome this issue, when encountering the case that states ”no references have changed”, the rule used to convert the generic `refs_after` on the left hand side should be used to convert this special case (except, instead of `refs_after` it is `refs_before`).

```
// refs_after(S<‘a> as TestTrait<’inner>, ‘inner) ->
↪  refs_after(S<‘a>, ‘a)
impl<'a, 'inner: 'a> TestTrait<'inner> for S<'a> {}

// refs_before(S<‘a>, ‘a) -> refs_before(S as
↪  TestTrait2<’inner>, ‘inner)
impl<'a: 'inner, 'inner> TestTrait2<'inner> for S<'a> {}

// refs_after(left as TestTrait<’c>, ‘c) = refs_before(left as
↪  TestTrait<’c>, ‘c) U refs_before(right as TestTrait2<’d>,
↪  ‘d)
fn foo<'a, 'b, 'c, 'd: 'c, T: TestTrait<'c>, U:
↪  TestTrait2<'d>>(left: &'a mut T, right: &'b U) {}

/* Calling foo with arguments so T is the concrete type S<'left>
↪  and U is of type S<'right>
refs_after(left as TestTrait<’c>, ‘c) := refs_before(left as
↪  TestTrait<’c>, ‘c) U refs_before(right as TestTrait2<’d>,
↪  ‘d)

apply refs_before(S<‘a>, ‘a) -> refs_before(S as
↪  TestTrait2<’inner>, ‘inner)
refs_after(left as TestTrait<’c>, ‘c) = refs_before(left as
↪  TestTrait<’c>, ‘c) U refs_before(right, ‘right)

apply the special case (refs_before version of the LHS) of no
↪  references changing
refs_after(S<‘a> as TestTrait<’inner>, ‘inner) ->
↪  refs_after(S<‘a>, ‘a) with s/after/before/g

refs_after(left as TestTrait<’c>, ‘c) = refs_before(left, ‘left)
↪  U refs_before(right, ‘a)

apply refs_after(S<‘a> as TestTrait<’inner>, ‘inner) ->
↪  refs_after(S<‘a>, ‘a) with left: S<'left>
refs_after(left, ‘left) = refs_before(left, ‘left) U
↪  refs_before(right, ‘a)

Monomorphised
*/
```

Figure 2.35: Full example monomorphising a generic function annotation using trait implementation annotations

Chapter 3

Implementation

3.1 Introduction

The project is split into 5 parts:

1. `dump-mir-optimised`: a tool to display the optimized MIR code for a program either in text format or as in the Graphviz dot language. Used to help development.

2. `rust-epcs`: the core library that calculates the EPCS from a MIR body

3. `rust-epcs-cli`: a simple tool to display the optimized MIR code together with the EPCS for each MIR statement calculated using the `rust-epcs` library. The output is either text or in the Graphviz dot language.

4. `lsp-epcs-hover`: a program that implements a small subset of the language server protocol (LSP) to respond to hover requests with the EPCS for that line in human readable format.

5. `vscode-epcs-hover`: a small Visual Studio Code extension to show the EPCS information of a certain line when the user hovers on it. This extension uses the `lsp-ecps-hover` application as the LSP server.

The meat of the thesis is contained in the `rust-epcs` library; the rest of the tools exist to aid development or to showcase how the EPCS information from the `rust-epcs` library can be used. All the programs, except for the `vscode-epcs-hover` extension, are written in Rust as they require access to internal Rust compiler data structures that cannot easily be accessed from any other language.

3.2 The `rust-epcs` library

3.2.1 Core Design

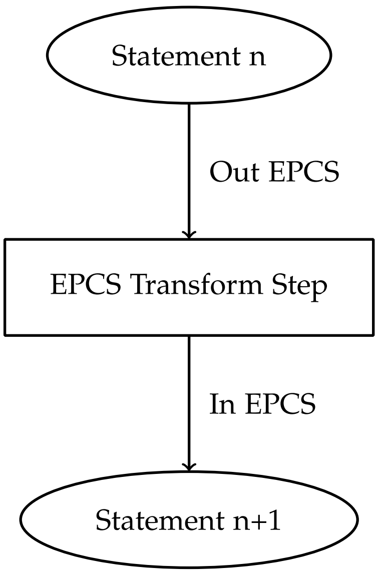
The principle of the `rust-epcs` library is that it takes a MIR body, which is the compiled MIR for a function body consisting of basic blocks and forming a control flow graph, and for each MIR statement in every basic block calculates the appropriate EPCS. The core principle that differentiates this implementation from the previous works in this field is that this implementation only uses Rust data structures that are deemed by the compiler team to be more stable, in this case optimized MIR. Previous implementations depended on unstable, and often changing, internal Rust data structures that are not meant to be used by external tools. While in theory based upon the previous standalone implementation[6] that accompanied the EPCS report , the vast majority of code has been written from scratch, with some core data structures being heavily modified rather than replaced.

The library has to iterate through the control flow graph, and for each MIR statement generate the resulting EPCS based upon the EPCS from the previous statement. Each basic block in the control flow graph has an EPCS before the block, and an EPCS after the block. The EPCS before a block is either empty, as for the first block, or is the result of merging all the EPCSs after every predecessor basic block. The EPCS after a block is the EPCS after the terminator statement of the block, or if there is no terminator, the EPCS after the final statement. To process a basic block the library calculates the EPCS before the block, uses this as the basis for calculating the EPCSs for all the statements in the basic block, and then sets the EPCS after the block to be the appropriate EPCS.

`rust-epcs` uses the data-flow analysis technique to iterate through the control flow graph. This technique means that the library iterates through the control flow graph, calculating EPCSs and following the edges, until the EPCSs calculated no longer change, i.e. a fix-point has been reached. If merging the EPCSs of the predecessor blocks has no effect on the EPCS before a block, then the block has reached a fix-point. Intuitively this is true because if the EPCS before the block has not changed compared to the previous iteration, then the EPCSs after each statement will not change as neither the EPCS before each statement has changed, nor have the statements themselves.

The use of approximations detailed in the theory chapter of this thesis means that iterating over cycles in the control flow graph will reach a fix-point.

To calculate the EPCSs for a MIR body the library uses a work queue, which is a queue of tasks that the library must perform. The calculation is only finished when the queue is empty.



There are three types of tasks that can be performed:

1. Process a statement.
2. Process a terminator of a basic block, this is the last statement that defines the successor blocks.
3. Merge the incoming EPCSs for a basic block to calculate the merged EPCS before the block.

Initially the library schedules all statements in the first block to be processed, as well as the terminator of the first block.

To process a statement the library takes the output EPCS of the previous statement, applies a transformation to the EPCS, and uses this transformed EPCS to calculate the EPCS after the statement.

Processing a terminator is much the same as processing a statement, except that it schedules work items to merge the incoming EPCSs for all successor blocks of this block. To calculate the new EPCS before a block the library merges all the incoming EPCSs. The library then compares the new EPCS before the block to the previous EPCS before the block, from the previous iteration. If the two EPCSs are the same then the block has reached a fix-point and nothing more is done. If the merged EPCS differs from the previous EPCS then the library schedules work items to process all the statements of this block and also schedules the terminator to be processed.

This use of the work queue allows the iteration over the control flow graph to be represented by three simple functions. It uses one central loop which fetches work from the work queue, rather than using complicated nested loops to iterate over all blocks and all statements.

The aforementioned transformation step is used to support cases where the format of the EPCS from the previous statement is different to the format required for the to-be-processed statement. A simple example is when accessing fields: if variable `x` is in the PCS but the next statement accesses `x.field1` there is the question about what the correct behaviour is. While it can be argued that having `x` in the PCS implies that all fields of `x` can also be accessed and no transformation need be applied, the decision was made to make this ″unpacking″ of fields explicit. In this context `x` will be removed from the PCS and replaced with all the fields of `x` (e.g `x.field1`, `x.field2`, etc). This decision to be strict regarding the variables in the PCS was partially taken to mirror the behaviour when ″packing″, which occurs in the opposite case. When the PCS includes all fields of a variable (e.g {`x.field1, x.field2, x.field3`}), but the current statement requires the use of the variable itself (e.g `x`), the library must make sure that *all* fields of `x` are in the PCS. To make it explicit that the requirement that all fields of `x` must be in the PCS to be allowed to use `x` is fulfilled, the transformation step ″packs″ the fields of the variable into the variable itself and replaces the fields of `x` in the PCS with `x` itself. Being strict and explicitly packing and unpacking also makes both manual and automatic checks that the implementation and EPCS is correct easier. The implementation currently asserts that every place that occurs in a MIR statement is also in the PCS in the exact same form.

Supporting more esoteric Rust cases, such as two-phase borrows, is made much easier through the use of the transformation step. Two-phase borrows occur in certain cases where there are conflicting overlapping borrows that don't lead to crashes but instead should be supported. A prime example is `v.push(v.len())`[5]. Figure 3.1 shows the MIR code generated by the compiler and that the mutable borrow of `tmp0` and `tmp1` overlap. . As a solution the developers of the Rust compiler introduced a special class of mutable borrows: sleeping mutable borrows. These sleeping mutable borrows do not impose their exclusivity constraints until they are used, when they are ″woken up″ and become normal mutable borrows. In figure 3.1 `tmp0` would be deemed a sleeping borrow that only enforces the mutable borrow constraints when used on line 4. Supporting these sleeping mutable borrows involves changing the transformation step to check if a sleeping borrow is used in the next statement, and to reflect the ″waking up″ of the sleeping borrow in the transformed PCS by changing the reference type to mutable.

```
tmp0 = &mut vec;       // mutable borrow starts here.. -+
tmp1 = &vec; // <-- shared borrow overlaps here         |
tmp2 = Vec::len(tmp1); //                               |
Vec::push(tmp0, tmp2); // <--.. and ends here-----------+
```

Figure 3.1: `v.push(v.len())` lowered to MIR showing how the borrows overlap.[5]

3.3 Theory and Implementation Comparison

3.3.1 Black Boxes and Wildcards

When calculating the EPCSs for a function body it is necessary to reason about what the function arguments, that are references, point to. With the example `fn foo(left: &mut T)` it is difficult to express what `left` points to, and therefore it is difficult to express the EPCSs for statements in the body of `foo`. At compile time it is known that `left` points to something of type `T`, and as it is a mutable reference `left` has exclusive access to whatever it points to. As the value of `*left` is unknown a way of expressing the case ”we know it points to something, and has exclusive access, but we don't know, and don't care exactly what” is needed. To this end some more theory is required, and so black boxes are introduced: `left -> BlackBoxArgument(1)`. `BlackBoxArgument(#argument index)` is a placeholder in the PCS to represent this case. .

Where `BlackBoxArgument` is used for arguments that are mutable references a different approach is needed for shared references. This is because shared borrows do not guarantee exclusive access to what they point to, and the possibility of aliasing needs to be represented. Aliasing allows the possibility that for `fn bar(one: &T, two: &T)`, `one` and `two` could point to the same instance of `T`. A new piece of notation is needed to represent aliasing, to represent that ”the shared borrow points to something but it does not have exclusive access”. To this end both `left` and `right` point to a ”wildcard”, e.g `left -> WildcardArg, right -> WildcardArg`. The use of `WildcardArg` encodes the fact that `left` and `right` could point to anything, including the same memory location.

There are also other cases where references point to things that are not variables in the program. One such case is the use of promoted constants, which occurs in MIR code when the block that calculates a constant has been ”promoted”, or elevated from the surrounding code and put into its own block. In the cases when promoted blocks return references or when the result is immediately borrowed before being assigned to a variable, the RCS must be able to express that a variable points to the result of a promoted constant block.

Rather than treat a promoted block as a function, which would create difficulties as the promoted block has no function signature, a new notation is needed to express ”the result of a promoted block number #id”: `BlackBoxPromotedConst(#id)`. This means that for `x = &mut PromotedConst[1]`, the RCS would include: `x -> BlackBoxPromotedConst(1)`. In the case where the promoted block returns references the type of the variable that the result is assigned to is used to apply the appropiate `BlackBoxPromotedConst` to the RCS.

Constants are not just promoted, they may also reside inline in the MIR. The most common example of this is the string literal `str`, which is usually immediately borrowed to make the type `&str`. To be able to express what the borrow `&str` points to another new notation is introduced: `BlackBoxInlineConst(#id)`, where the id must be unique for that inline constant (a simple counter is enough to satisfy the uniqueness requirement). Used in context, if `x = &"Neither a borrower nor a lender be"`, then the RCS will include `x -> BlackBoxInlineConst(1)`.

3.3.2 Compiler lifetime information

When designing the theory it was assumed that there would be easy access to the lifetime information and lifetime relationship information in the places where these relationships are defined. Examples include function signatures (`fn foo<'a: 'b, 'b>`) or structure definitions (`struct T<'a: 'b, 'b`). This information is easily accessible at the higher HIR level: the abstract syntax tree first gets lowered to HIR, then lowered again down to MIR. However in the process of lowering the HIR the lifetime relationship information - the outlives information - is not put into the MIR but is converted into a format appropriate for the borrow checker.

It does not appear to be easily possible to extract the precise lifetime information needed from the borrow checker data structures. This is because the borrow checker deals with many implicit lifetime relationships, as well as the explicit lifetime relationships that the theory uses, mixing these two types of lifetime relationship together. At the point in the Rust compiler pipeline where optimised MIR is generated, the borrow checker has erased the lifetime information from variable type information.

When analysing function signatures to generate function reference annotations the lifetime information of the argument types has not been erased, however it is still not possible to easily access outlives information. The lack of outlives information means that currently function signatures with lifetime relationships, such as

```
fn foo<'a: 'b, 'b>(left: &'a T, right: &'a T) -> &'b T}
```

are unsupported, but functions that only use one lifetime are supported

```
fn foo<'a>(left: &'a T, right: &'a T) -> &'a T
```

The lack of lifetime information in types means that if the RCS includes a `refs(x, 'a)` statement, and `x` has more than one lifetime parameter it is impossible to determine which references in x `refs(x, 'a)` actually refers to. This makes it difficult to support references in data types that do not support split borrows, such as enumerations and slices, as the theory requires this lifetime information. A workaround implemented by the `rust-epcs` library exploits the fact that lifetime information for types is available when processing function signatures. This makes it possible, when applying a FRA to a PCS, to replace `refs(x, 'a)` with a definite set of all fields in `x` with a lifetime outliving `'a`.

3.3.3 Unimplemented theory

The `rust-epcs` library implements the `reachable` approximation in order to make sure that the EPCSs calculated by iterating over the control flow graph reach a fix-point.

The `refs(x, 'a)` approximation, as mentioned before, is only used after the application of a function reference annotation to the EPCS. Approximating references in non-split borrow types, such as enumerations and slices, the use of `refs` is not currently supported in the implementation due to the aforementioned erasure of lifetimes.

The implementation of function reference annotation rules is limited by the lack of lifetime outlives information. However the `rust-epcs` currently implements a limited version of rule 1 and 2, which only support functions that use one lifetime instead of relating lifetimes. Rules 3, 4 and 5 are currently not implemented, nor are traits.

3.4 Extension VSCode

To help demonstrate the real world use of the `rust-epcs` library a Visual Studio Code extension was developed. The extension, consisting of the `lsp-epcs-hover` and `vscode-epcs-hover`, shows a human readable EPCS when hovering over a line of Rust code. It uses a mapping between MIR statements and file line numbers to find the most appropriate MIR statement and accompanying EPCS, and then turns the EPCS into a more human readable form. This involves looking up the Rust variable names for MIR local places, and converting field indexes and variant indexes to their human readable names.

Chapter 4

Evaluation

4.1 Theory Design

Developing the additional theory in the thesis allowed the implementation to move away from using internal Rust compiler information. This move has several benefits, first of all it means that upgrading the compiler version is much easier as the theory relies on a more stable interface rather than the unstable and often changing nll-facts.

Part of the reason for using the more stable optimized MIR was to see if optimized MIR exposed enough information to be able to calculate the EPCS information. The conclusion is that easy access to some important lifetime information is missing, as stated above in subsection 3.3.2, however the vast majority of required information is available. Basing the theory underpinning the implementation only on the MIR output, and not relying on internal Polonius information, would allow, if the compiler exposed a stable form of MIR without borrow checking it, for `rust-epcs` to explain why programs that do not borrow check are incorrect. Therefore the theory enables the implementation to no longer have to write and read the internal compiler borrow checking information to and from disk, resulting in a more performant implementation.

The approximations `reachable` and `refs` are successful because they follow the approximations that the compiler uses. This was expressly done to avoid inconsistencies between `rust-epcs` and the compiler. Especially with the `refs` on types that do not support split borrows, it is possible to be more precise than the compiler. However this would mean that the EPCS could explain why a program is correct but the compiler rejects the program. If the compiler decides to support split borrowing on slices or enumerations then they can be treated in the same way as types that do support split borrows.
.

The function reference annotations were designed as framework for describing relations, and a list of rules which create these relations. The reason for this design was that it was assumed that not *all* function cases would be covered, so a way to easily add new rules, and allow the implementation to cleanly implement a subset of theory was needed. To simplify the implementation not all rules need to be implemented, as rules can be added at a later date without having to modify existing code.

As function reference annotations can be expressed in a human readable form, they can be used to educate newer Rust users as to the effects of functions on references.

The notation used for function annotations was validated by the ease of extending the theory to generics and traits, which were not considered when first designing the notation.

4.2 Completeness

There are two ways in which the implementation lacks support for Rust code. The first is where the implementation panics when encountering an unsupported type, the second is where the implementation requires some extra logic to identify and handle cases. Many of the unsupported Rust cases are not particularly interesting from a theory standpoint, and hence were not prioritised. For example MIR repeat statements are the equivalent of just assigning each field, which is supported.

A non-exhaustive list of unsupported Rust cases:

- External crates
- Processing function bodies where the arguments are data structures with references inside
- Slices
- Ascribed user types
- MIR aggregate statements
- Asynchronous MIR terminators
- Repeat MIR statements
- Raw pointers
- Unique and shallow borrows
- `drop_replace` terminators
- Function arguments that are functions

- MIR nullary operands
- References in types that do not support split borrows
- Lifetime relations, e.g. `'a: 'b`
- Function reference annotations rule 3, 4 and 5
- Function reference annotations for traits
- Some uses of `Box`

4.3 Implementation

4.3.1 Design

The data-flow analysis technique used to iterate through the control flow graph is perfectly suited to the task of calculating EPCSs, as it addresses the fact that cycles in the control flow graph make it necessary to calculate the EPCSs for a block multiple times until the input EPCSs of a block have stabilised to a fix-point.

Using a work queue to represent the tasks required for the data-flow analysis greatly simplified the EPCS builder; the alternative would be many nested loops to iterate over all blocks and statements multiple times, or an implementation that uses recursion to visit the control flow graph nodes. Both of these alternative techniques would lead to suboptimal code and complicate debugging, especially with the complicated stack traces of a recursive implementation.

The results of the `rust-epcs` library are exposed in a format similar to the format used by Prusti for its MIR analysis steps, which will simplify the integration of the `rust-epcs` library with Prusti. This result format is simple, yet contains all the EPCS information for before and after every MIR block and MIR statement. This allows tools such as the Visual Studio Code extension to easily process the EPCSs for a MIR body and transform them into a human readable format.

The result format uses only one very simple MIR data structure to link an EPCS with its location in the MIR body. This allows tools to quickly give the EPCS result context with the correct MIR statement. All other data types are defined by the `rust-epcs` library. This makes the result format, and therefore tools that use `rust-epcs` less affected by any potential changes in Rust compiler data structures.

The library also offers a simple external interface for tools, with only one method call required to calculate the EPCS on a MIR body and type context (TyCtxt) object. Apart from this simple external interface the `rust-epcs` offers a number of helpful functions for tools, including the pretty-printing of EPCS statements where variable names, field indices and variants are replaced by their human readable names. The Visual Studio Code extension demonstrates how these helper functions allow a useful tool to be implemented in very few lines of code.

Using a transformation step in between the output EPCS of one statement and the input of another greatly simplifies adding support for two-phase borrows to the `rust-epcs` library. Despite the fact that the implementation was not consciously designed to support sleeping mutable borrows, it is a testament to the flexibility of the implementation and the principle of an intermediate transformation step that adding two-phase borrows required changing very few existing lines of code.

To increase confidence that the output of the library is correct the implementation includes numerous assert statements. For example, before a place is used, it is asserted that it exists in the EPCS. Developing the implementation with these assertions helped uncover numerous bugs and inconsistencies without needing to manually check the resulting EPCSs. This saved time and was less error prone then solely relying on manual comparing.

As the theory for function reference annotations does not cover every single case, such as passing closures to functions, the decision was made to have a list of rules that are run to calculate the function reference annotations. These hot-pluggable rules mean that the implementation can very easily extended to cover these more complex function signatures. When the theory takes a leap forward it will be painless to implement these new function annotation rules. The framework that runs all the rules and merges the results into one function annotation already exists.

4.3.2 Comparison to previous implementation

The current implementation, like the previous implementation, provides a simple command line application to give an example of how the library is used. The previous implementation does not output any results at all. In stark contrast the current implementation outputs a colourful Graphviz representation of the control flow graph with MIR statements and terminators annotated with the relevant EPCS before and after. This Graphviz representation not only serves as an example for how to interact with `rust-epcs` results but also aids development by providing a very intuitive representation of the library output, making it easier to manually check that the EPCSs are correct.

Unlike the previous implementation, the current implementation does not use any forked Polonius code, which was required to expose some internal Polonius data structures. Not forking the Polonius code removes a maintenance burden, and makes it much easier to update the implementation to a newer compiler version, as the Polonius code must change in lock-step with the compiler. The current implementation also uses the more stable form of MIR: optimised MIR . Using this rarely changed form of MIR, and no private internal data structures, means that there will be fewer issues when updating the compiler version that `rust-epcs` supports. The previous implementation supports the nightly compiler version from 2019-12-27, and when updating to 2020-07-27 the format of the `nll-facts` and the MIR had changed. This meant that the Polonius fork had to be updated and the code that visited the MIR had to be changed too.

4.4 Performance

The previous works in this field used unstable internal Rust data structures, non-lexical lifetime facts (`nll-facts`), to help calculate borrow information. To access the `nll-facts` information the facts are first written to disk by the Rust compiler and then immediately read by the previous implementations fork of Polonius to calculate borrow information. Not only is the disk used to transfer internal information, but the `nll-facts` written to disk contain more information than needed, as the data is designed for a more general use-case. This further reduces the efficiency of the previous implementation. Using the disk as a method for transferring internal data is inherently slow and could lead to disk performance becoming a bottleneck in tools such as Prusti when analysing complex programs. To avoid this bottleneck the `rust-epcs` implementation in this thesis calculates the borrowing information directly from the optimized MIR, with no intermediate reading or writing to disk.

To show how these differing approaches affect run-time a series of benchmarks were constructed. Care was taken to avoid common pitfalls:

- Each implementation was compiled using `cargo build --release` to let the Rust compiler apply optimisations.
- Before each execution of an implementation the implementation executable and benchmark case were copied to a new temporary directory. This means that any data from the previous benchmark runs of that benchmark case are not cached and do not skew the times. Especially for the previous implementation where the `nll-facts` were written to disk, it is important for the integrity of the times that it does not use a cached version of the `nll-facts` but instead generates them again.
- Both implementations were modified to prevent the printing of intermediate and final results. Printing information to stdout or stderr takes a surprisingly long time, especially if the benchmark cases are very long. In extreme cases the current implementation was 4× slower than the previous implementation until it was noticed that this was

due to the current implementation printing intermediate and final results and the previous implementation was not.

- The previous implementation was modified to no longer panic when it found Rust cases that it did not support, which occurred in many of the benchmark cases. Instead the previous implementation skipped over these cases; this will make the previous implementation score slightly higher in the benchmarks than it would if it calculated the exact same information as the current implementation. However it was deemed that the difference would not be substantial enough to invalidate the benchmarks and fixing the previous implementation would be a distraction from implementing the current implementation.

- The Rust compiler versions are different; the previous implementation uses a nightly compiler from 2019-12-27 and the current implementation uses a nightly compiler from 2020-07-27. The compile times for the benchmark programs have decreased in the newer compiler version in comparison to the older version. To account for the decrease in compile times figure 4.2 shows a graph which shows the median benchmark time minus the median compile time for the appropriate compiler version. Both compilers were run with the same compiler flags

  ```
  -Zalways-encode-mir -Zmir-opt-level=0 --cap-lints=allow
   ↪  --crate-type=lib
  ```

  A noticeable absence is that `-Znll-facts` is not a given flag, because passing it causes compile times to increase. As the `nll-facts` are only required by the previous implementation and it contains some of the information calculated by the current implementation it was deemed generating the `nll-facts` should count towards the time taken to calculate EPCSs.

- Before calculating the EPCS the Rust compiler must first compile the code, this compile step introduces systematic error to the benchmark timings. Compiling the Rust code also has its own random error. Due to the random and systematic error introduced by first needing to compile the benchmark cases, the benchmarking script took multiple samples to reduce the impact these errors had on the results.

First the test cases used by the `rust-epcs` library as integration tests were run by the current implementation and the previous implementation. However, not much information was gleaned from these benchmark cases, as the time to calculate the EPCSs paled in comparison to the time the Rust compiler took to compile the benchmark cases, which was roughly the same in both cases. Clearly small programs with a linear or almost linear control flow would not differentiate the implementations as the vast majority of time was spent compiling.

To this end two benchmarks were designed which should test the worst-case performance of the implementations, making them ideal as benchmarks. Loops require fix-point iterations which, especially with nested loops, will become very time intensive. The first benchmark involved nested while loops that iterated through a linked list. The nested depth of these while loops were varied from 0 to 550, in intervals of 50, and for each nest depth 500 samples were taken to help deal with systematic and random timing errors introduced by the Rust compiler. As the graph in figure 4.1 shows, as the number of nested loops increases, and the time taken to compile contributes less to the overall time taken, it becomes clear that the current implementation has significant performance benefits over the previous implementation.

The second benchmark is similar to the first benchmark, except that it uses nested for loops. In the same way as the first benchmark the depth of nested for loops were varied from 0 to 200, in depth increments of 10, and for each increment 100 samples were taken. As the graph in figure 4.3 shows, as the number of nested loops increases, and the time taken to compile contributes less to the overall time taken, it becomes clear that the current implementation again has significant performance benefits over the previous implementation.

The two graphs show that in the worst performance cases the current implementation is between 2$\times$ and 3$\times$ faster than the previous implementation. However the current implementation was not created with performance as the utmost priority and therefore there is still some low hanging fruit that can be optimised in order to further improve the performance. One example of this is that intermediate calculations are currently not cached. In cases where the calculations change only slightly they should be cached and modified rather than being calculated again from scratch as currently occurs. The data structures that hold the EPCS information could be optimised for certain cases by greater use of hash maps to avoid expensive lookups.

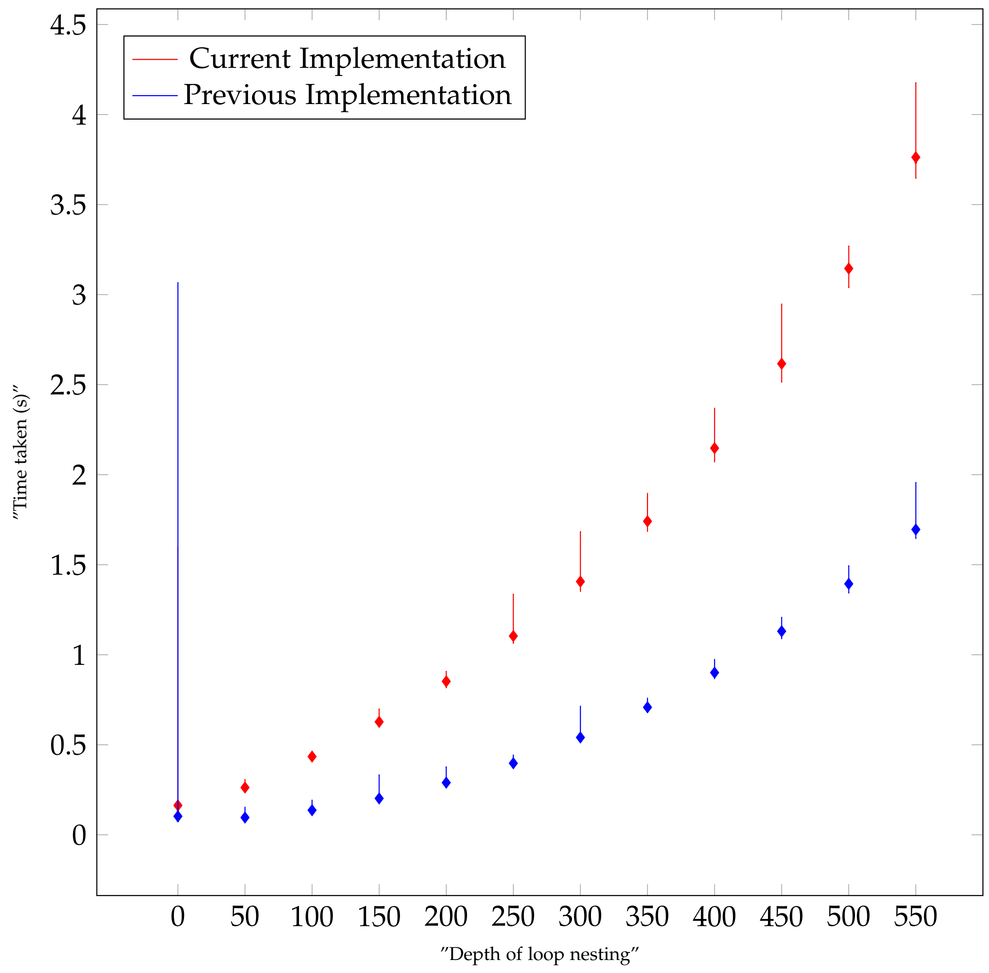


Figure 4.1: Graph showing the runtime of the current and previous implementation when calculating EPCSs for nested `while` loops of various depths

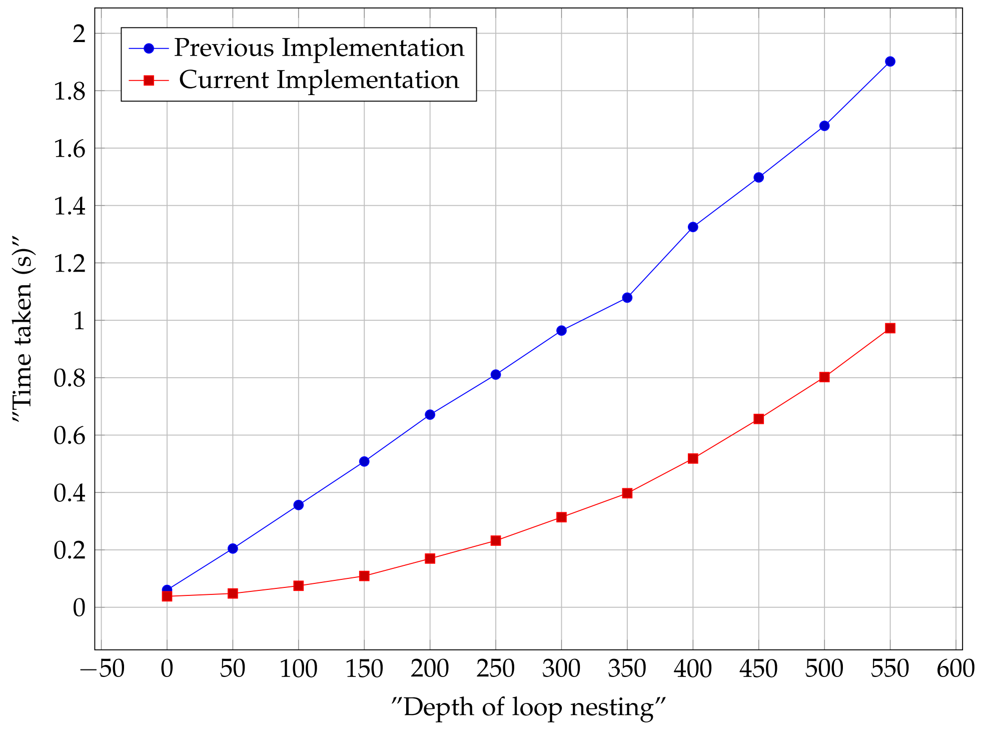


Figure 4.2: Graph showing the runtime of the current and previous implementation minus the median time taken to compile when calculating EPCSs for nested `while` loops of various depths

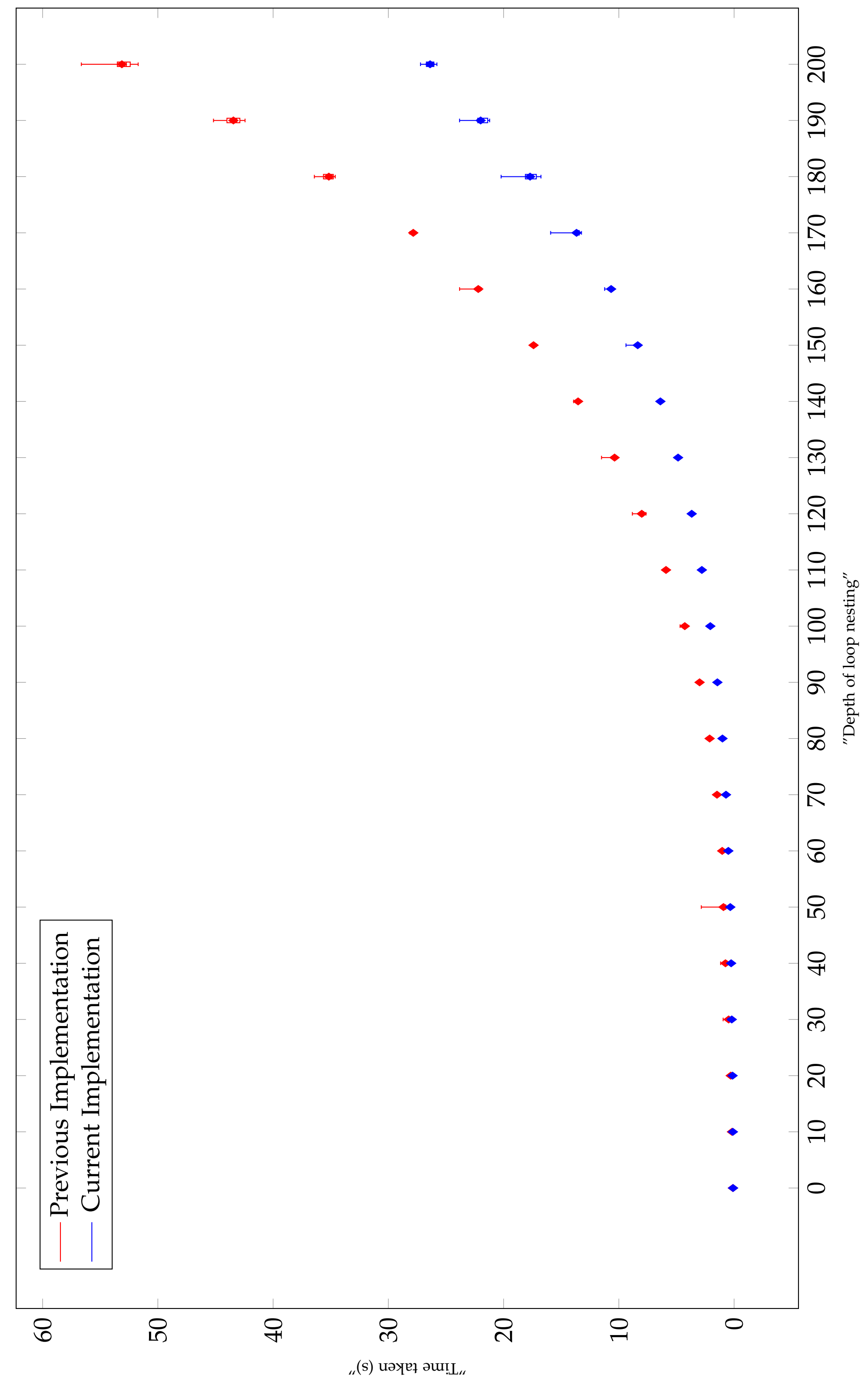


Figure 4.3: Graph showing the runtime of the current and previous implementation when calculating EPCSs for nested `for` loops of various depths

4.5 Future Work

There is scope for future work.

The function reference annotation intuition for traits should be expanded to become rules. More complex trait cases and passing functions to functions should be examined and rules for generating function reference annotations for these cases should be developed.

The implementation would benefit from some refactoring and tidying up. The core data structures, and the methods used to manipulate the core data structures were designed before all uses of the data structures were clear. Now that the access patterns and the data formats required for calculating EPCSs without using Polonius information are well known, there are various changes that should be made to reduce code reuse and improve performance.

Naturally the implementation should also be extended to cover the Rust cases that it currently does not support.

To increase confidence that the output of the implementation is correct a further project could be to compare the results from this implementation to internal borrow checker information over a wide range of programs. If combined with increasing the range of Rust code that the implementation supports the wide range of open source Rust programs could be used as a test set.

Another avenue for exploration would be to create tools that use this EPCS information, such as a fuzzer that uses EPCS information to be able to more intelligently generate correct or subtly incorrect Rust programs.

Chapter 5

Conclusion

In this thesis we designed theory to be able to calculate EPCSs over a variety of Rust cases without relying on an external oracle. Much of this theory was then implemented in a standalone tool that calculates the EPCSs for a bigger subset of Rust programs than the previous implementation. Additionally, the Visual Studio Code extension which displays human readable EPCSs, demonstrated that the implementation was fit for purpose and made the results more accessible. Also adding support for two-phase borrows showed the flexibility of the implementation and the ability to quickly add support for complex and unforeseen Rust cases.

As the evaluation states the theory enables the implementation to be between $2\times$ and $3\times$ faster, while not using unstable internal Polonius data. Possible future work has been described, in order to improve the implementation and to extend the theory to cover complex function calls involving traits and closures. This thesis and the accompanying implementation allow the creation of next generation Rust tooling that uses fine-grained type information to be able to reason about Rust programs in detail without the use of internal borrow checker information.

Bibliography

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 3, pages 147:1–147:30. ACM, 2019.

[2] The Rust Book Contributors. Rust book chapter 4.

[3] The Rust Project Developers. Polonius, 2020. Accessed September 08, 2020.

[4] N. Matsakis. Introducing mir, 2016.

[5] N. Matsakis. Nested method calls via two-phase borrowing, 2017.

[6] Dylan Wolff. Extended place capabilities summaries for rust programs. Research in Computer Science project, ETH Zürich, 2019.

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):

Computing fine-grained type information for Rust programs

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

Name(s):	**First name(s):**
Seddon	William

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	**Signature(s)**
Zürich, 06.09.2020	[illegible]

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