0 Summary

In this thesis we explore borrow-checking compiler-phase of the programming language Rust. Borrow-checking is a method of ensuring memory safety in Rust; that is the absence of dangling pointers, data races, and references to invalid data among others. A value 42 has one owner x, in a Rust statement such as `let x = 42;`. The variable x will always be the owner of the value but other parts of a program may need to access the value of x. Lending out only read or both read and write-access are described as immutable and mutable access, respectively. The borrow-checker phase of the Rust compiler attempts to ensure that all loans of an owners associated value are in congruence of a simple set of rules: at any time, many immutable loans may exist or one mutable loan may exist, and the owner must live longer than any loan of its value.

Rice's theorem states; all interesting questions about the input/output behaviour of programs are undecidable [24]. Hence any program analysis, and therefore also the borrow-checker, must be approximate. We would likely want such an analysis to always answer correctly in the rejecting case, such that we catch all problematic programs but this leaves us with false negatives: programs which the borrow-checker rejects but are in fact legal wrt. the stipulated borrowing-rules. This is the motivation of this thesis: how to improve borrow-checker precision in Rust?

Research into crafting borrow-checkers are sparse but work exists in [23] on formalising subsets of Rust into a calculus which enforces both type- and borrow-safety without the notion of the borrow-checkers; Non-Lexical Lifetimes or Polonius. The work presents soundness proofs and quasi-mechanical proof by way of model checking 500B programs.

We develop our own borrow-checking algorithm inspired by the inability of Non-Lexical Lifetimes (NLL) to handle the `get_or_insert.rs`program and the context that NLL provides; that is lifetimes of borrows and their potential collision with other conflicting borrows.

This work is in conjunction with a proof-of-concept implementation of our algorithm; `mir-owner-guillotine`. Our proof-of-concept is able to parse and construct our own Intermediate Representation (IR) for a subset of the Rust Mid-level Intermediate Representation (MIR). On our IR we implement common compiler analysis algorithms; reaching definitions analysis and liveness analysis. Finally, we implement our borrow-checking algorithm on top of the built IR and liveness results to ensure that no borrows lifetimes are intersecting when not allowed according to the Rust rules of borrowing.

We evaluate our proof-of-concept on two Rust programs which NLL currently rejects and on a trivially borrow-wise invalid program. For NLL-rejected programs we define an ad-hoc but promising method of obtaining MIR to test our implemented analysis with. Furthermore, we present arguments of correctness but no formal proof for our analysis, which is delegated for future work.

We believe the field of borrow-checking is an interesting and under-explored domain within language formalisation and verification which warrants further investigation.

On Borrow-Checking Analysis Precision in Rust

Falke Bjernemose Øtker Carlsen

Cassiopeia, Department of Computer Science, Aalborg University, Denmark
fvejlb17@student.aau.dk

Abstract

We investigate the precision of existing borrow-checking analyses; the current Non-Lexical Lifetimes (NLL) and future Polonius. This is motivated by the exemplary program `get_or_insert.rs`, which is rejected by NLL, as it employs seemingly ordinary patterns that users might expect to function.

We aim to understand the current and future borrow-checkers of Rust and propose our own borrow-checking analysis on Rusts Mid-level Intermediate Representation, based on the classic compiler liveness analysis and encoding Rusts borrowing rules into overlapping borrow-expression. With our proposed analysis follows a proof-of-concept implementation in Python which passes two accept cases currently rejected by NLL and rejects a trivially invalid program.

Out results are rebutted by unproven translation of NLL-rejected Rust programs into MIR and lack of correctness proof for our analysis.

1 Motivation

Borrow-checking is an analysis-technique to ensure memory safety and data integrity by avoiding dangling pointers and race conditions. Unfortunately, this analysis is difficult to perform on large, complex codebases.

Rust; a newer systems programming-language has risen in usage, and likeability by users in the last decade, one again taking first-place as most loved programming language according to Stack Overflow's yearly developer survey [21]. Unlike garbage-collected languages like Java, Python, and Go where memory is deallocated automatically - and sometimes unpredictably - Rust takes the approach of ownership and borrowing to safely manage memory.

Ownership for Rust is defined as three rules: "Each value in Rust has an owner, there can only be one owner at a time, and when the owner goes out of scope, the value will be dropped" [9]. Assigning a variable to another variable copies the value if this is of a compile-time known fixed type, if it is dynamically sized the value is stored on the heap and an assignment as before performs a shallow-copy and additionally invalidates the first variable - this is called a move.

Borrowing in Rust is the method for which values are lent out to be temporarily used. Instead of pointers from languages such as C, Rust uses references and guarantees that these point to valid memory of some type, unlike C. These references are either read-only, or read-write called immutable and mutable, respectively. To avoid data races, whenever a mutable reference to some value is live, no simultaneous references to the value other than that reference may exist. [9]

While ownership and borrowing as concepts has been proposed for programming language memory safety in work such as [4], Rust is the first *mainstream* programming language employing it. Therefore, this work will look into Rust's application of the concept.

2 Introduction

Rusts default borrow-checker as of August 2022 [16] is called Non-Lexical Lifetimes (NLL), which famously rejects the following, seemingly innocuous, Rust program:

```
1 fn get_or_insert(map: &mut HashMap<u32, String>) ->
  ↪  &String {
2  match HashMap::get_mut(&*map, &42) {
3   Some(v) => v,
4   None => {
5    map.insert(42, String::from("init"));
6    &map[&42]
7   }
8  }
9 }
```

Code block 1: `get_or_insert.rs` [15]

The function in code block 1 takes a mutable map as an argument in its signature, and checks if a value exists for some index in the match-statement in line 2. A match-statement in Rust is a pattern match, `HashMap::get_mut()` returns a value of Enum-type `Result` which is either `Some(v)` with the given value `v`, or `None()` with no value. If a value exists at that index, we take the Some-branch in line 3 and immediately return it. If not we take the None-branch, insert a value at the index at line 5, and immediately return the freshly inserted value at line 6. Omitting semi-colons for statements as seen in lines 3 and 6 is Rust syntactical sugar for implicit return.

Rust strives to allow all known-safe programs to compile. However, there are patterns a user will expect to be allowed, which are rejected due to borrow-checker imprecision. These restrictions can be consequences of borrow-checker implementation, e.g. performance and complexity.

The `get_or_insert.rs`-program above in code block 1 is disallowed by the Rust compiler today because NLL believes the returned variable `v` which is a reference into the map, is also live at the insertion of a default value into the map. If this is true, then we have both a mutable and immutable reference to the map at the same time which is disallowed by the Rust borrowing rules [9]. However, we see that this is a false negative judgement of NLL, since only one match-branch may execute for any invocation of the method.

Exactly this imprecision of NLL for a pattern which we would naively expect to be accepted is the motivation for investigating borrow-checking analysis precision in Rust. The false negative for the `get_or_insert.rs`-program is a known limitation of NLL described by the NLL RFC2094 [15], which in turn has spawned interest in more advanced borrow-checkers such as Polonius, conceived in blog post [13] by Niko Matsakis, also a main contributor in the development of NLL. Polonius can handle the pattern in code block 1 but Polonius itself is still a work-in-progress.

In this thesis we will explore how a borrow-checker for Rust may be developed which handles the exemplary program `get_or_insert.rs`, which NLL rejects. We begin with establishing required compiler theory, Rust peculiarities, the NLL and Polonius borrow-checker, and related work in Section 3. In Section 4 we define our approach to borrow-checking in Rust. Section 5 will discuss implementation of our borrow-checker and its required parser and Intermediate Representation (IR). We present and position our results in Section 6. Lastly, in Section 8, we will conclude on the findings of our work.

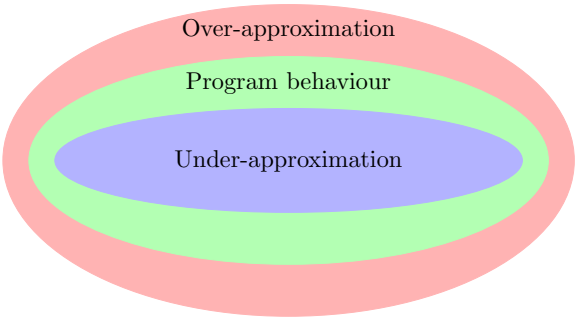


Figure 1: Approximations of program behaviour. Areas denote the amount of behaviour an approximation could cover.

3 Background

We will explore the background for borrow-checking in Rust and position borrow-checking among related work. We will also provide relevant theory required for describing our analysis.

3.1 Rust peculiarities

Rust introduces more memory safety with its memory model with the concepts of ownership and borrowing, such as no dangling pointers to invalid data, concurrent safety by borrowings effect on data-races, and performance by richer information on the use of values. [9] However, the borrow-checker can never obtain perfect accuracy; rejecting all faulty programs and accepting all correct programs. Rice's theorem states: *"all interesting questions about the input/output behaviour of programs are undecidable"* [24]. We can however obtain approximate answers to program properties. These approximate answers are more useful, if they are sound, meaning the analyser may respond with 'maybe' if unsure but for either 'yes' or 'no', the answer must be correct. [18]. A borrow-checker should always be conservative, s.t. it rejects wrong programs. This is an over-approximation of the behaviour actual program behaviour. This relationship is illustrated in Figure 1.

```
1 let x = 5;
2 let raw_ptr: *const i32 = &x as *const i32;
3 let y = unsafe { *raw_ptr };
```

Code block 2: Dereferencing a raw pointer by use of `unsafe`

Therefore Rust introduces the `unsafe`-keyword, to disable the borrow-checker for the `unsafe` scope as seen in code block 2. This code compiles but may

exhibit undefined behaviour, e.g. if dereferencing a raw pointer, which is normally not valid.

Using `unsafe` is required for operations which the compiler analysis would always reject but the developer attempted ensure is correct, such as implementing a linked-list [20].

Rust also has a concept of *moving* ownership from one variable to another, as briefly introduced in Section 1. This can be made explicit by the keyword `move`. For primitive stack allocated types, assignments do not force a move of ownership as these types often implement the Copy-trait. However for heap allocated memory this is the default behaviour as copying the value from the original variable can be computationally expensive. [7]

3.2 Dataflow analysis

Dataflow analysis reasons about the flow of data in a program on AST or CFG IR-structures.

3.2.1 CFG

A CFG is a graph with vertices containing a sequence of statements, vertices contain basic blocks. Basic blocks are a sequence of statements with only one entry and exit point. Directional edges imply possible flow between basic blocks.

$$CFG = V, E$$

V is a list of vertices

E is an edgelist, (source, target)

3.3 Mid-level Intermediate Representation

For Rust to adopt NLL borrow-checking, a suitable IR for dataflow-analysis had to be established; Mid-level Intermediate Representation (MIR). MIR is broadly a collection of functions with type-declarations, for which their body is represented as a CFG containing vertices of sequential statements with flow controlled by built-in primitive statements.

For the purposes of borrow-checking, we can omit much of the information MIR contains and therefore, we will only discuss the points which are of interest for our analysis.

For the following simple Rust program in code block 3, we can obtain its corresponding textual MIR by using compiler-tools such as rust-playground [6].

```rust
fn main(flag: bool) {
    let mut a = 1;
    if (flag) {
        a = 0;
    } else {
        a = 42;
    }
    let b = a;
}
```

Code block 3: Rust program with control-flow

Rust-playground can output the textual MIR for the above Rust program, which is only meant for human consumption. This is however the most accessible way to achieve MIR for Rust programs, since it does not require intimate knowledge of the Rust compiler which we due to time-constraints were not able to achieve.

```
fn main(_1: bool) -> () {
    debug flag => _1;
    let mut _0: ();
    let mut _2: i32;
    let mut _3: bool;

    bb0: {
        _2 = const 1_i32;
        _3 = _1;
        switchInt(move _3) -> [false: bb2,
        ↪  otherwise: bb1];
    }

    bb1: {
        _2 = const 0_i32;
        goto -> bb3;
    }

    bb2: {
        _2 = const 42_i32;
        goto -> bb3;
    }

    bb3: {
        _4 = _2;
        return;
    }
}
```

Code block 4: Simplified MIR for Rust in code block 3

Statements in MIR are assignments to locations or terminators such as `goto`, `unreachable`, `return`, etc. Locations represent a path a memory location.

In MIR the CFG is encoded as a sequence of blocks named `bb`$n-1$ for n nodes. The edges are represented with MIR statements `goto, switchInt, return` among others. The first block `bb0` is an implicit entry into the function. Continuing on, when we discuss CFGs in the context of MIR analysis, we refer to the CFG described by the textual MIR representation as shown in Figure 4.

MIR is on static single-assignment form (SSA), however, when variables can have multiple values, it uses phi-nodes [1] to merge two control-flow paths resulting in potential multiple assignments to one location. This is seen by the assignment of either 0 or 42 to location 2, in blocks 1 and 2.

3.4 Phi-function

Programs on static-single assignment (SSA) form has only one definitions of variables. This is useful for simplifying and optimising dataflow analysis algorithms [1].

However, when two control-flow branches merge, we no longer have only one assignment to each variable. Therefore, we use *phi*-functions so that we can combine multiple possible assignments:

$$\phi(v) = \begin{cases} v_1 & \text{if control}_1 \\ v_2 & \text{if control}_2 \\ \vdots & \vdots \\ v_n & \text{if control}_n \end{cases} \tag{1}$$

Here we yield the v_n for which control-condition is satisfied. The control-condition *picks* the variable which was assigned in the given control-flow.

3.4.1 Liveness analysis

Liveness is a classic compiler-analysis which informs on at which points in the program a given variable is *live*. A variable is live, if its current value is needed at a later point. [1]

We can intuitively say that every use of a variable *generates* liveness and a definition *kills* liveness of a given variable. We need to map statements to the DEF- and USE-sets of variables which the statement uses. For our analysis, we will operate on MIR as discussed in Section 3.3, below we show the intuitively expected location sets of DEF and USE for two MIR-statements from `get_or_insert.mir`:

$$DEF([\texttt{\_1 = const 1\_i32;}]) = \{1\} \tag{2}$$

$$USE([\texttt{\_2 = HashMap::get(\_3, \_5)}]) = \{3, 5\} \tag{3}$$

We need to define the DEF and USE equations for all statements in MIR to properly describe their semantics. Unfortunately, formal semantics exist for neither Rust or MIR, so we base our equations on documentation in [14] and our best judgement but note that we present no formal proof that the following definitions are correct wrt. MIR behaviour.

We introduce the utility function $uses(expr_{MIR}) \rightarrow Locs$ which returns all locations $\{l \mid l \in CFG\}$ for which they are present in expression $expr_{MIR}$. Recall from Section 3.3, that we with CFG refer to the CFG that MIR describes.

For an expression that contains a function-call, we only return the locations in the function-call arguments if they are moved, this is syntactically written in MIR by the `move` keyword before a location, e.g. `switchInt(move _1);`. We only include moved locations as the `move` keyword signifies changing ownership of value from owner to consumer as described in Section 3.1. This is caused by the move invalidating the original owner as part of the mechanism in Rust which avoids dangling pointers, such that only the new owner may use or borrow out the value.

The *uses* function is defined in Equation 4 where $args$ are the set of arguments to a function-call and $call(a)$ is a condition which is true if the expression contains a function-call and an expression a as arguments. Both the simple expression e and arguments a from $call(a)$ are syntactical fragments of MIR.

$$uses(e) = \begin{cases} \{l \mid l \in e \wedge \neg call(a)\} & \\ \{l \mid l \in call(a) \wedge call(a)\} & \text{if } l \text{ is moved} \\ \emptyset & \text{otherwise} \end{cases} \tag{4}$$

In the following, we misuse expected syntax of DEF and USE by directly referencing statements in the CFG, the correct and more verbose way would be $CFG[n] = stmt_{MIR}$ and $USE(n) = \{42\}$. For all assignment statements in MIR we define the DEF- and USE- sets in Equations 5 and 6.

$$DEF([loc_d = E;]) = loc_d \tag{5}$$

$$USE([loc_d = E;]) = uses(E) \tag{6}$$

MIR has an implicit use of location 0 for its `return;`, so we define its USE as shown in Equation 7.

$$USE([\texttt{return;}]) = \{0\} \tag{7}$$

For liveness, we define $OUT[n], IN[n]$ as shown in [1]:

$$OUT[n] = \bigcup_{s \in succ[n]} IN[s] \tag{8}$$

$$IN[n] = USE[n] \cup (OUT[n] - DEF[n]) \tag{9}$$

For all statements n in the CFG, we initialise $IN[n], OUT[n] = \emptyset$.

Computation is backwards s.t. a use of a location generates liveness, and definition of a location kills it. We compute iteratively until a fixed point is reached as shown in Algorithm 1.

Algorithm 1: Liveness analysis from [1]

input : cfg, program CFG
output: IN, OUT, array of computed sets

```
foreach stmt n ∈ cfg do
    IN[n] ← ∅;
    OUT[n] ← ∅;
end
repeat
    foreach stmt n ∈ cfg do
        IN'[n] ← IN[n];
        OUT'[n] ← OUT[n];
        OUT ← ⋃_{s∈succ[n]} IN[s];
        IN[n] ←
          USE[n] ∪ (OUT[n] − DEF[n]);
    end
until IN'[n] = IN[n] ∧ OUT'[n] =
  OUT[n] foreach n;
```

Upon completion of Algorithm 1 we obtain the array of live IN- and OUT-sets for all program points in the program.

3.5 Non-Lexical Lifetimes Borrow-Checking

In 2017 RFC 2094 [15] proposes a new borrow-checker supporting non-lexical lifetimes.

We introduce a simple, classic borrow-checker error in code block 5 and formally explain the approach of both NLL and Polonius based on documentation available [17, 13, 8, 15].

```
1 let mut x: u32 = 42; // mutable u32 int
2 let y: & u32 = & x; // reference to x
3 x += 1; // use (mutate) of x
4 println!("{}", y); // use of y
```

Code block 5: Classic borrow-checker error program, due to use while borrowed [17].

The issue with the Rust program in code block 5 is that while `x` is borrowed by `y` we try to mutate it at line 3, which becomes an issue when it is later used at line 4 by printing `y`.

We get an error at some **statement** N if:

- the statement N accesses a path P
- and accessing the path P would violate the terms of some loan L
- and the loan L is live

A **path** P is an expression that leads to a memory location to which we can assign, such as:

- `x` a variable is a memory location on the stack
- `x.f` a field of a path is a memory location
- `*x.f` by a pointer at field `f` from variable `x`
- `(*x.f[_])` some index into an array

A **loan** is the name for a borrow expression, e.g: `&x`. A loan has an associated path P and a mode for how it is borrowed; mutably or immutably.

The rule for an immutable loan of some path P is P must not be modified, only read from the newly created reference or from a subsequent immutable loan of the path P. Similarly, for a mutable loan of some path P, P must not be accessed in any way except for the reference created by the borrow expression.

Recall liveness from Section 3.4.1: A loan is live if the reference that it created - or a derived reference from it - may be used later.

With NLL, Rust computes **lifetimes** for every reference, to decide for which set of nodes in the MIR CFG that reference might be used. These lifetimes are at times syntactically present in Rust source code to assist the borrow-checker or render better error-messages; e.g. `'a`, by explicitly assigning a name to a lifetime of some reference. We will substitute the lifetime variable for the actual set of nodes where the loan is live, e.g. $'a \rightarrow \{\texttt{1,3,4}\}$.

For presenting lifetimes, briefly consider the line numbers of the classic borrow-checker error program in code block 5 to be the indices of the CFG

nodes for the following. The computed, explicit lifetimes are not more formally described than 'something that might be used later' in [15, 17]. The lifetimes seem to correspond with our $IN[n]$ definition from Section 3.4.1 as they refer to what variables are live before the *execution* of a statement. We now annotate the program with explicit lifetimes:

```
1 let mut x: u32 = 42;
2 let y: &'a u32 = &'b x;
3 x += 1;
4 println!("{}", y);
```

Code block 6: Program in code block 5 annotated with lifetimes `'a` and `'b`

We see that `y` is live on lines 3, 4 since it will be used by line 4 and thus lives between its declaration at line 2 until that point, therefore we have `'a` $= \{3, 4\}$. Additionally, all the lines for which `y` is live, must include the lifetime of `y`.

However, we have a subtyping-rule; if lifetime `'i` flows into `'j`, then `'i` must outlive `'j`, because we cannot have a reference to some data for which the reference lives longer than the data as that would leave a dangling reference.

Therefore, we are constrained on the lifetime of `'b`, and this results in `x` having to outlive `y`, and the lifetime `'b` gaining those CFG nodes: `'b` $\cup$ `'a` $= \{3, 4\}$. Note that lifetimes for each reference is in turn is the lifetime of its loan.

Now we have the required information to borrow-check the program according to the definition of an error at a statement N. Line 3 modifies the path `x` (N accesses a path P). By modifying the path `x`, we violate the terms of loan `&x` in line 2, and the loan with lifetime `'b` is live on lines 3 and 4. Thus we have a borrow-checking error at this statement with this context.

Using MIR makes development of the borrow-checker easier, since development can happen on the natural data structure for data-flow analysis as explained in this section. This higher precision allows for and requires more fine-grained definitions of the borrowing-rules. The second motivation point for developing MIR, behind simplification and desugaring, is aiding the borrow-checker phase [14].

Recall the prime example of NLL's shortcoming; `get_or_insert.rs` from code block 1:

```
1 fn get_or_insert<'a>(map: &'a
  ↪  mut HashMap<u32, String>) -> &'a String {
2  match HashMap::get(&*map, &42) {
3    Some(v) => v,
4    None => {
5      map.insert(42, String::from("init"));
6      &map[&42]
7    }
8  }
9 }
```

Code block 7: `get_or_insert.rs` annotated with syntactical lifetimes

NLL rejects the program in code block 7, because the named in-function-signature lifetime of `map`; the syntactical fragment `<'a>` according to NLL must last to at least the end of the function, and importantly; across all possible codepaths, even those unreachable when the borrow starts. This restriction is a consequence of the implementation of NLL as we will explore later.

Therefore, NLL rejects the program and reports that the mutation of `map` at line 7 happens while a borrow due to the `Some(v) => v` is still live. However, we can clearly see that the branches in the two cases, either some value exists or it does not, are entirely *disjoint* in terms of borrows: Either a value is present in `map` and it is returned, or we insert a default, and return it all the same.

In [17] it is explained that the NLL development group had an approach to capture this type of borrow-checker imprecision but it was computationally expensive. In that effort it was discovered that there existed more complex patterns which NLL could not handle, which lead to the development of Polonius.

3.6 Polonius Borrow-Checking

Polonius argues that *growing* lifetimes of borrows in a forward manner is not the best approach. Instead Polonius searches for the *origin* of each reference R, which is a set of loans which R might have originated and hence backwards. The following section is based on [13, 17, 8].

We repeat the program annotated with lifetimes from code block 6 in code block 8 to explain Polonius.

```
1 let mut x: u32 = 42;
2 let y: &'a u32 = &'b x;
3 x += 1;
4 println!("{}", y);
```

Code block 8: Program in code block 5 annotated with lifetimes `'a` and `'b`

We still need inference as with NLL in Section 3.5 but starting with the origin variable `'b` because we are working in the other direction. We no longer infer the set of lines for which the variable is live, rather the set of loans. The origin of `'b` is itself; loan $\{L1\}$ because it was created right at its use.

We assign to `y` once which is again loan $\{L1\}$. Had we seen more assignments to `y`, we would instead take the union of all the loans generated by the assignment to`y`. This leaves us with the conclusion that all references originates in the `&x` expression and we have computed all the origins of this small program.

Note how computing origins did not require the use of lifetimes, only the dataflow relationship of 'when we create a reference, where does it get stored to'.

Polonius slightly modifies the third rule of when we might get an error at some program statement N. Recall from Section 3.5: We get an error at some program **statement** N if:

- the statement N accesses a path P
- and accessing the path P would violate the terms of some loan L
- a loan L is live if **some live variable** has L in its type

The change in Polonius is that we look at the liveness of variables wrt. their types as highlighted by the emphasised rule three above.

Consider the program in code-block 8 for which we see that line 3 modifies the path x which is violates the terms of loan $L1$, much in the same fashion as NLL in Section 3.5.

Now we need to look at which variables are live at this line 3 program point. We see that variable `y` is live because it will be used by line 4 and its type is `y: &{L1} u32` which includes the loan $L1$. This causes us to register an error, due to accessing to a path x for which a loan $L1$ exists violated by this access, and $L1$ being live because of later use of the same loan $L1$ by another reference `y`.

With NLL we directly compute the lifetimes of references without a particular error in mind. This is in contrast to Polonius which computes origins and only uses liveness of loans through reference to figure out if we might have a conflict.

3.7 Related work

We will look at related research using static analysis to verify properties in Rust. The majority of current research focuses on improving memory safety at runtime and not explicitly crafting or formalising a better borrow-checker.

In [20], a static taint analysis is implemented for MIR. To accomplish this goal, the authors define operational semantics for a subset of MIR, notably omitting types and references. The work contributes a formalisation of MIR sufficient for taint analysis, defines a taint analysis on their semantics, and implement the analysis as a Rust tool.

In [5], the authors applies instrumentation to the defined semantics from [20] with the goal of verifying whether Rusts borrowing system lends benefits to static analysis. Additionally, a theoretical approach to reduce the state space of taint-analysis is presented.

In [2] Prusti is presented; a static analysis tool for specifying and verifying Rust program are absent of panics, also suited for overflow checks and, while remaining usable for mainstream developers without formal verification domain knowledge. Prusti encodes capability information - adjacent to the borrowing-concept - into implicit dynamic frames logic [25], a similar logic to separation logic [22]. Prusti needs to know precise knowledge of the capabilities at any program point for verification, especially framing. Authors define an algorithm to compute precise summaries of the capabilities held at each program point, called 'place capability sets' (PCS). These sets are required for the 'core proof', when encoding the program, specifications into authors intermediate verification language Viper [19] being a heap-based imperative language with pre- and post-conditions, and loop-invariants.

Similarly to separation logic, Viper enforces that a field location can only be accessed when permission is held to do so, while this is held it cannot change by others providing framing and argues the need for PCS and capabilities. Viper field permissions are tracked in the program state as affine resources; they can be explicitly added or removed from state, or implicitly dropped if not required. Each Rust memory location is mapped to a Viper field location.

Prusti encodes this capability information for verifying program behaviour generally, not specifically for borrow-checking.

In [12] Lindner et. al. presents a symbolic execution analysis utilising the KLEE LLVM execution

engine [3]. By adapting KLEE engine to analyse LLVM bitcode from Rust programs, the authors can statically ensure memory safety and panic-free execution of Rust code by symbolic execution.

The analysis requires writing pre- and post-conditions for the program being analysed, and contracts (P, f, Q) P, where P, Q are first-order logic predicates being Boolean expressions on a, b for $f : a \to b$. These are given to KLEE's SMT solver and is still subject to path explosion; 2^c for c conditionals.

Programs checked by this work can be moved from safe to unsafe-rust, to improve performance and the authors argue the programs are still safe and simpler, since no panic handling is needed. There is no direct contribution to the accuracy of borrow-checking, yet the work complements borrow-checking by removing parts of a program out of borrow-checking responsibility into unsafe.

In [23] Pearce presents 'Featherweight Rust'; a lightweight formalism for Rust which captures both the flow-insensitive type checker and the flow-sensitive borrow-checking analysis which enforces the ownership invariants. A reference implementation in Java with which Pearce has model-checked the calculus using over 500 billion input programs resulting in one confirmed compiler bug and other lesser issues found.

In [11], Li et. al. presents MirChecker, an automated static analysis tool for bug-detection. By use of numerical and symbolic information for its static analysis to detect runtime panics and memory-safety errors by constraint solving techniques. Most of the found bugs are not memory-safety bugs, but triggers of runtime panics.

4 Borrow-checking analysis

In this section we will describe our analysis, argue it's correctness, and explain required setup to have `mir-owner-guillotine` analyse Rust programs.

4.1 Intuition

The intuition behind our analysis is that over-approximation to all code-paths are unnecessarily conservative, especially for the prime example of `get_or_insert.rs` in Code Block 1, since we know the two scopes in the match-statement are wrt. borrowing, disjoint.

We therefore seek to show that the variable `v` cannot be live, whenever we are in the scope which mutates the map.

Figure 2 illustrates a simplified CFG of `get_or_insert.mir` with sequential nodes compressed into one for brevity, based on `get_or_insert.mir`-construction as shown in Appendix C for which we will include details in the following Section. The match-node signifies the borrow of `&mut map`, the Some- and None-arms the body of each case, and ϕ-Some, ϕ-None nodes the merging of possible return value.

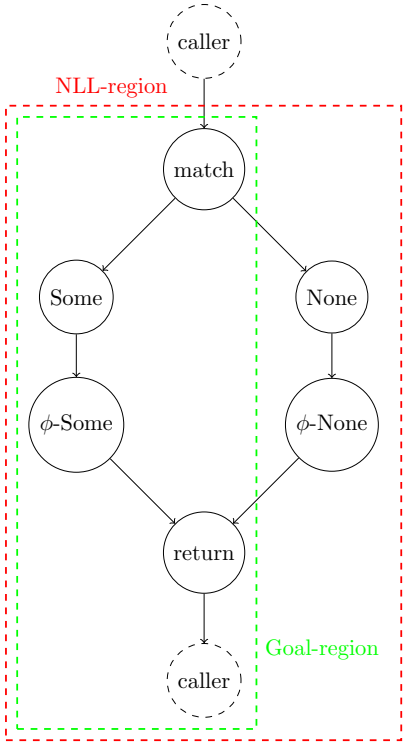


Figure 2: Simplified CFG of `get_or_insert.mir` with `v`-live regions

Recall from Section 3.5 that NLL disallows `get_or_insert.rs` because the lifetime of `v` grows to encapsulate every code-path from the match-statement and to, at least, the function caller. In Figure 2, this region is shown as the red NLL-region denotes the span of code where NLL considers `v` live. On the other hand, we would like for `v` to only be live for the green goal-region.

We will use the insertion of the ϕ-nodes to reason that the control-flow paths which do not include the ϕ-node of a branch, ensures the locations used by that branch are not live. From this follows that we can simply use liveness to solve the imprecision of NLL borrow-check in the `get_or_insert.rs`-case.

4.2 MIR construction

The process of analysing MIR for Rust programs which NLL rejects is hindered, in that the compiler halts before textual MIR is available for the user. This necessitates an alternative procedure for

obtaining MIR representing our NLL-reject cases. This issue was also identified in [20].

The preferred method for obtaining this MIR would likely be to modify the borrow-checking phase of the Rust compiler to allow rejected code. One approach could be to wrap problematic code spans in the `unsafe`-keyword which disables the borrow-checker for that scope.

Unfortunately, rust-playground [6] does not seem to disable the borrow-checker no matter which scopes are wrapped in `unsafe`-scopes and we are not able to enable provisional Polonius features which would allow the program to be accepted.

We opt to use a hand-crafted method of obtaining MIR for NLL-rejected Rust programs:

1. Copy Rust program into MIR-A and MIR-B
2. Mutate control-flow paths of MIR-A and MIR-B such that at least one path retains original intent
3. Extract basic blocks from MIR-B encapsulating NLL-offending statements after compiling with rust-playground
4. Merge extracted blocks into MIR-A
5. Remove artefacts from mutation in step 2, correcting block-numbering collisions, and shift control-flow statements accordingly

The correctness of our ad-hoc hand-crafted method hinges on the fact that our substitutions does not affect the NLL-offending statements which we extract in step 3 above and that our reconstruction of the MIR is semantically equivalent to the expected MIR program.

We have not found other feasible methods of obtaining MIR for NLL-rejected programs than our method described here. We argue that the changes we introduce are slight and so we expect the resulting MIR to be in many ways similar to actual Rust compiler MIR.

Unfortunately, we cannot guarantee that this method is sufficient. This diminishes the application of any of our NLL-reject case results and warrants a high priority in future work.

We believe the necessity of using our analysis on NLL-rejected MIR warrants the use of our method.

Recall the `get_or_insert.rs`-example from code block 1:

```
1 fn get_or_insert(map: &mut HashMap<u32, String>) ->
  ↪  &String {
2  match HashMap::get_mut(&*map, &42) {
3    Some(v) => v,
4    None => {
5      map.insert(42, String::from("init"));
6      &map[&42]
7    }
8  }
9 }
```

To convince the compiler to output MIR, we could for MIR-A and MIR-B comment out line 5 and replace line 3 with `Some(v) => &map[&42],` respectively. By this, we obtain a MIR-A case where we no longer insert into the map in the None-case, resulting in no borrow of `map` while NLL believes `v` is live. In the other case, returning `&map[&42]` in the Some-case, while still a reference into the map, the reference is no longer generated by the match-statement, which NLL accepts.

This method produces `get_or_insert.mir`and a loop-cond-mut program based on the Polonius test suite. Finally, a handwritten trivially borrowing-wise invalid MIR program is written for the negative case. These and their construction are described in detail in Appendices C, D, and E, respectively.

4.3 Phi-nodes

Recall phi-functions from Section 3.4, used for consolidating multiple possible values of a conflicting variable when merging control-flow branches.

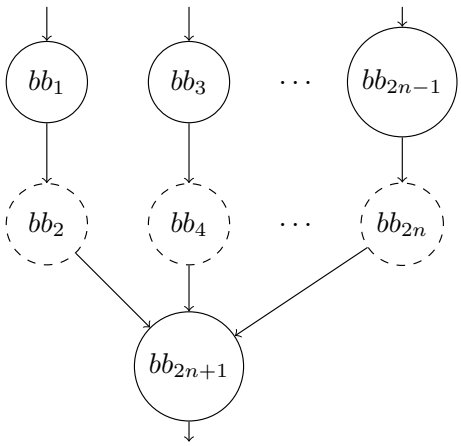


Figure 3: Hypothetical CFG illustration of insertion of dashed phi-nodes for n merging branches

```
1  bb0: {
2      _1 = param(_99); // stmt to simulate param
3      _4 = &(*_1); // conditional control flow
4      _10 = discriminant(_4);
5      switchInt(move _10) -> [0_i32: bb1, 1_i32:
       ↪  bb3];
6  }
7  bb1: {
8      _7 = SomeOperation(_1);
9      goto -> bb2;
10 }
11 bb2: {
12     _0 = _7;
13     goto -> bb5;
14 }
15 bb3: {
16     _2 = &mut (*_1);
17     _42 = SomeOperation(_2);
18     goto -> bb4;
19 }
20 bb4: {
21     _0 = _42;
22     goto -> bb5;
23 }
24 bb5: {
25     return;
26 }
```

Code block 9: Hypothetical MIR hand-written to show similar behaviour to CFG in Figure 3 example on phi-nodes, where shared variable is location 1

In MIR ϕ-nodes are inserted after each branch, assigning the conflicting variable into a shared location as illustrated by Figure 3 and shown in code block 9. We start with a conditional and setup in block 0, lines 1-6, which are omitted from the CFG figure. If b_1 places the conflicting variable in location 7 (line 8), and b_2 places the same in location 42 (line 17), then the subgraph could be written in MIR as shown in code block 9 for two merging branches. MIR typically uses location `_0` for return values.

For a conditional branching in MIR with n cases, and c branches that assigns conflicting values to the same variable, we would have c phi-nodes inserted between the last block of each c branch and the following block where control-flow merges after branching. Blocks for which no conflicting variables are assigned, e.g. `unreachable` or a block which does not change the value, are not appended phi-nodes.

Consider the hypothetical MIR in code block 9; if location 1 is a mutable resource NLL would reject the originating program, if the control-flow statement that leads to either block 1 or block 3 also uses location 1. This is problem-case 3 as described in NLL RFC 2094 [15].

However, for each trace the use of the conflicting location 1 which results in a borrow, is *killed* by the ϕ-node since it by assignment uses the location, rendering liveness able to limit the lifetime of the borrow to that branch only. We must consider and argue whether this is intended behaviour of the analysis, as we will not present a formal proof for our analysis.

In Figure 4 three subgraphs are presented for liveness analysis; a trivial sequential case (a), a conditional case (b), and a case with an infinite loop (c). These subgraphs are hypothetical and are constructed based on our understanding of MIR CFGs presented in Section 3.3 and 3.4. The naming is as follows; c is used for a common or control-flow dependant variable, c_n are values derived from c, v signifies a value that branches compute which are conflicting upon merging control-flow. The loop case has dotted and dashed back-edges from $t7$ signifying two possible loops back before branching, instead of returning v we simulate a use of v with our own keyword `use`. We initialise all states' IN- and OUT-sets to $\emptyset$ as described in Section 3.4.1.

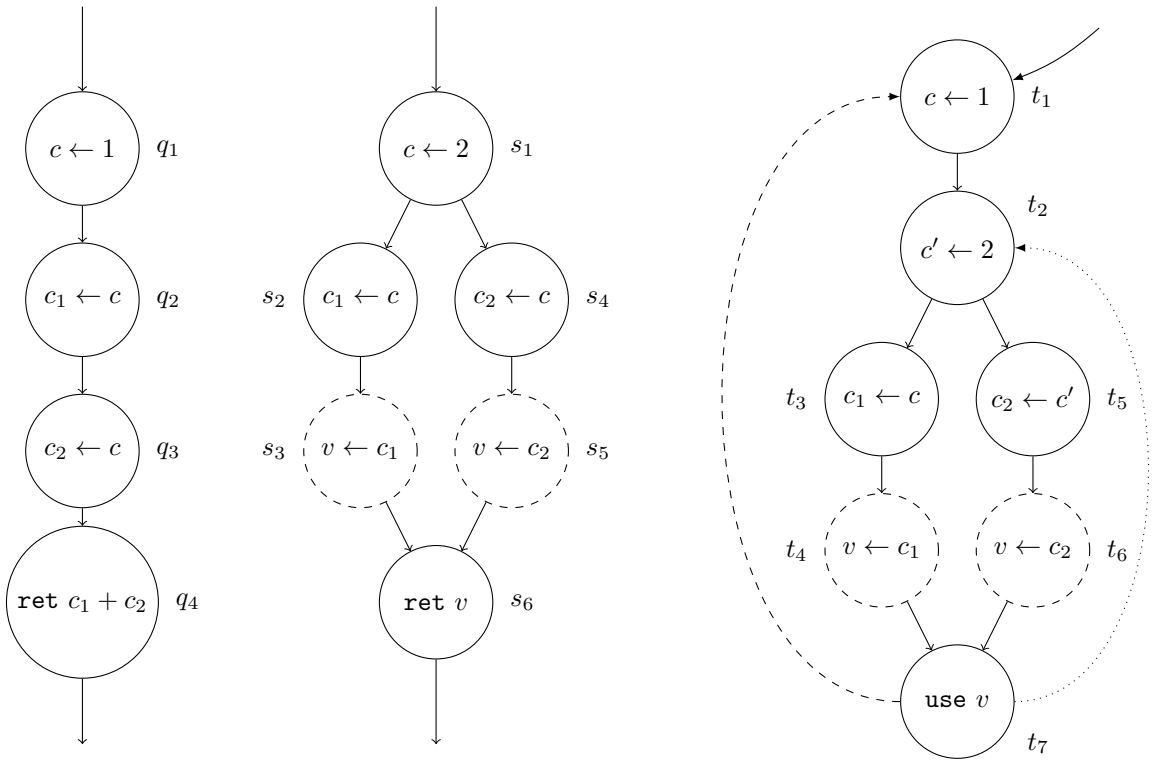
For the linear case we see the results tabulated in Figure 5(a). Recall that liveness analysis is backwards; against the flow. In q_4 we have uses of both c_1 and c_2 and union them to $IN(q_4)$. In q_3 we see a definition of c_2 and therefore we kill its liveness, but since we have not defined c_1 yet, we get $IN(q_3) = \{c_1\}$.

In the conditional case in Figure 4(b) and results in Figure 5(b) we compute liveness much in the same way as the linear case. Recall the definition $OUT[n] = \bigcup_{s \in succ[n]} IN[s]$ in Section 3.4.1. The $OUT[n]$ definition applied to node s_1 renders $OUT[s_1] = IN[s_2] \cup IN[s_4] = \{c\}$ signifying the only variables passed on to the branches is c. For node s_6 we use the variable v, which the phi-nodes s_3 and s_5 define. This renders us with a *closed scope* of liveness in the conditional case because of the phi-nodes and importantly wrt. the `get_or_insert.rs`-example we see that neither c_1 nor c_2 are live in their opposite branches.

4.4 Loops

Cyclical paths in the CFG could be problematic because we may no longer be able to reason as to when a variable is dead. Fortunately, the phi-nodes ensures this is not the case for our analysis. Consider the loop-case subgraph in Figure 4(c) and its result table in Figure 5.

The results for the loop-case show that $OUT(t_7) = \emptyset$, meaning we carry no liveness



(a) Linear case (b) Conditional case, ϕ-nodes dashed (c) Loop case, ϕ-nodes dashed

Figure 4: CFG subgraphs for liveness analysis

	$LIVE_{IN}$	$LIVE_{OUT}$
q_1	$\emptyset$	$\{c\}$
q_2	$\{c\}$	$\{c_1\}$
q_3	$\{c_1\}$	$\{c_1, c_2\}$
q_4	$\{c_1, c_2\}$	$\emptyset$

(a) Linear case

	$LIVE_{IN}$	$LIVE_{OUT}$
s_1	$\emptyset$	$\{c\}$
s_2	$\{c\}$	$\{c_1\}$
s_3	$\{c_1\}$	$\{v\}$
s_4	$\{c\}$	$\{c_2\}$
s_5	$\{c_2\}$	$\{v\}$
s_6	$\{v\}$	$\emptyset$

(b) Conditional case

	$LIVE_{IN}$	$LIVE_{OUT}$
t_1	$\emptyset$	$\{c\}$
t_2	$\{c\}$	$\{c, c'\}$
t_3	$\{c\}$	$\{c_1\}$
t_4	$\{c_1\}$	$\{v\}$
t_5	$\{c'\}$	$\{c_2\}$
t_6	$\{c_2\}$	$\{v\}$
t_7	$\{v\}$	$\emptyset$

(c) Loop case

Figure 5: Liveness results for subgraphs in Figure 4

through when looping back to the definition of either c or c'. Similarly, we generate liveness for c and c' only at t_1 and t_2. This leaves us with a case reminiscent of the conditional-case in Figure 4(b); neither c_1 nor c_2 are live in their opposite branches.

4.5 Identifying errors

Our borrow-checking algorithm relies on the fact that conflicting variables keeping borrows alive for longer than required is mitigated by ϕ-nodes *killing* liveness by merging the value into a common location. This happens

However, we need to identify borrows first which is handled by our borrow computation algorithm, shown in Algorithm 2. A borrow is defined in `mir-owner-guillotine` as a triple of the left-hand side location, the borrower, and the right-hand side location, the borrowee, along with a flag for whether this is a mutable borrow.

Algorithm 2: Borrow computation

input : cfg, program CFG
output: borrows: list of borrows

$b_s \leftarrow \{\}$;
$b_f \leftarrow \{\}$;
foreach *stmt* $n \in cfg$ **do**
  if *n is a simple borrowing stmt* **then**
    $b_s \leftarrow b_s \cup \{(n.l_{lhs}, n.l_{rhs}, n.mut)\}$
end
foreach *function-stmt* $n \in cfg$ **do**
  if *n is a function using a borrowed location* **then**
    $b_f \leftarrow b_f \cup \{(n.l_{lhs}, n.l_{rhs}, n.mut)\}$
end
return $b_s \cup b_f$

Algorithm 2 collects all borrows in the program, both simple borrows such as `_2 = &_1;` and more complex *reborrows* such as `_4 = HashMap::get(move _3);` such that we carry the borrowing of location 3 into location 4. This information would otherwise be lost due to the SSA form of MIR.

The function *overlaps* is defined as shown in Equation 10 and returns true if given borrow-locations c_1, c_2 are live initially for any statement in the CFG.

$$overlaps(c_1, c_2) = \begin{cases} true & \text{if } \forall n \in CFG: \\ & c_1 \in IN[n] \wedge \\ & c_2 \in IN[n] \\ false & \text{otherwise} \end{cases} \quad (10)$$

Algorithm 3: Borrow-checking

input : borrows: list of borrows
output: valid: boolean of borrow-check

foreach set s of borrows for borrowee b **do**
  if $|\{x \mid x \in s \wedge s.mut\}| \geq 1$ **then**
    foreach combination c_1 and c_2 of s **do**
      if $\neg c_1.mut \wedge \neg c_2.mut$ **then**
        continue
      end
      if $overlaps(c_1, c_2) \wedge (c_1.mut \vee c_2.mut)$ **then**
        return $false$
      end
    end
  end
end
return $true$

The heart of our borrow-checking analysis is shown in Algorithm 3. We take the computed borrows from Algorithm 2 and split them into tuples of a borrowee location and sets of borrows for this location: $(loc, \{Borrows\})$.

For each combination of two borrows of a borrowee, we check our following rules which are based on Rusts borrowing rules [9]:

1. If c_1 and c_2 are immutable, we continue, as two immutable references allowed.
2. If at least one of (c_1, c_2) are mutable and both borrower-locations ($c_n.l_{lhs}$) are live at the same time, we report an error.

5 Implementation

We implement our analysis in a self-contained tool; `mir-owner-guillotine`[1] written in Python. Using `sly`, a parser-generator library for Python, we lex and parse MIR into our own IR for easier analysis later on. The grammar we construct to recognise a subset of the MIR language is found in Appendix A and is constructed for our purposes, this also due to no grammar being available from MIR.

While MIR is already an IR which is almost on Single Static Assignment form [14], we chose to make our own IR to more easily work with the program in a form that we control. Basic blocks have their own data-class with all concomitant data stored locally. Edges are represented in edge-list format, and statements are typed to make analysis

[1] `github.com/falkecarlsen/mir-owner-guillotine`

more concise. Both blocks and statements have appended sets of predecessors and successors. We also choose to ignore information contained in MIR, which is not relevant for our purposes and reduces unnecessary complexity but are for code-generation in later Rust compiler-stages.

We only care about types in-so-far as methods on them dictate specific borrow-semantics:

```
_2 = HashMap::<u32, String>::get::<u32>(move _3,
↪  move _5);
```

Above we see a MIR statement from the `get_or_insert.mir` program. `HashMap::get` returns a Option-type, which is an enum which is either `Some(v)` or `None()`. The method gets, by move-semantics, location `_3` and `_5`, which signifies they are used and therefore not live after the statement.

However, the Result-type, now stored in location `_2`, may be read by e.g. `_7 = discriminant(_2);` without being consumed, since the discriminant-function only reads whether a value is present, i.e. it is of type `Some(v)` or `None()`. Only later in the example program, is `_2` consumed while unwrapping by casting to a Some-type:

```
_8 = ((_2 as Some).0: &std::string::String);
```

In the above, we do not care about the types of values, beyond their dataflow interaction required for analysis, since previous stages of the Rust-compiler has already done type-checking and we therefore may assume the program is correct with respect to these completed stages. The definitions of IN- and OUT-equations in Section 3.4.1 establish how MIR statements generate and kills liveness of locations.

Appendix B shows a conceptual UML diagram of our IR. The largest unit is a `CFG`-class containing entry- and exit-nodes, basic blocks, and their edges. A `CFG` also has methods for computing reaching-definitions, liveness, and our borrow-checking.

A `BasicBlock`-class has a list of all statements within in, along with live locations in and out of each statement.

Similarly, a `Statement`-class, has all concomitant data on it, methods for generating its definitions and uses, whether it borrows. Depending on the type of `Statement`, it overrides some of these methods, as e.g. `FunctionStatements` may reborrow and uses locations differently. We also include `PrimitiveFunctionStatements` to handle MIR's non-assignment primitives; `goto, return, unreachable` among others.

5.1 Liveness analysis

We need liveness analysis to know whether locations are live for potential conflicting borrows. The implemented algorithm in `mir-owner-guillotine` is similar to Algorithm 1 in Section 3.4.1.

We annotate each statement with the concrete set of IN and OUT along with each node in the CFG with its IN and OUT, corresponding to its first and last sequential statement, respectively. `mir-owner-guillotine` also outputs tabulated results of the computed liveness of each location.

5.2 Borrow-checking

Our analysis culminates in our borrow-checking algorithm based on Algorithm 3 in Section 4.5. The implemented algorithm in `mir-owner-guillotine` additionally reports error-locations in the CFG by the tuple $(bb_n, stmt_n)$ signifying where in the MIR program an error is found.

In `mir-owner-guillotine` we output each check with its context of borrows currently checked and reports the result of each of our defined borrowing-rules, based on Rusts own described borrowing rules in [9].

6 Discussion

The implemented `mir-owner-guillotine` borrow-checking analysis accepts the `get_or_insert.mir` presented in Appendix C, accepts the loop-condmut program from the Polonius test suite [10] presented in Appendix D and rejects the trivially borrowing-wise faulty MIR program in Appendix E as expected.

We argue that we capture the essence of the NLL-problem for `get_or_insert.rs` in our borrow-checker, implemented in `mir-owner-guillotine`. Consider the borrows we identify in `get_or_insert.mir`below. A borrow is a five-tuple defined as $(borrower_{loc}, borrowee_{loc}, mutable, bb_n, stmt_n)$. Where bb_n and $stmt_n$ denotes the indices in the MIR program where the borrow occurs.

- Borrow(er=3, ee=1, mut=False, bb0, s1)
- Borrow(er=2, ee=1, mut=False, bb0, s3)
- Borrow(er=11, ee=1, mut=False, bb7, s0)
- Borrow(er=17, ee=1, mut=True, bb2, s0)

Recall for the final time `get_or_insert.rs`:

```
fn get_or_insert(map: &mut HashMap<u32, String>) ->
↪  &String {
 match HashMap::get_mut(&*map, &42) {
   Some(v) => v,
   None => {
     map.insert(42, String::from("init"));
     &map[&42]
   }
 }
}
```

For which we register concurrent liveness for any combination of borrows for which at least one of them are mutable borrows. The relevant are locations 3, 2, 17, and 11. Location 3 signifies an immutable reference to map used in the match-statement, generated by `&*map` in line 2. Location 2 signifies the resulting value of the call to `HashMap::get_mut(&*map, &42)` in line 2, specifically the Option-type explained in Section 2. Location 17 signifies the mutable reference taken to insert default value into map, corresponding to line 5. Location 11 signifies the None-branch return of newly inserted default value, because of last statement at line 6.

```
Bcking borrowee 1 with borrows:
        Borrow(er=3, ee=1, mut=False, bb0, s1)
        Borrow(er=2, ee=1, mut=False, bb0, s3)
        Borrow(er=11, ee=1, mut=False, bb7, s0)
        Borrow(er=17, ee=1, mut=True, bb2, s0)
...
checking           Borrow(er=3, ee=1, mut=False)
and                 Borrow(er=17, ee=1, mut=True)
no overlap
...
checking           Borrow(er=2, ee=1, mut=False)
and                 Borrow(er=17, ee=1, mut=True)
no overlap

checking           Borrow(er=11, ee=1, mut=False)
and                 Borrow(er=17, ee=1, mut=True)
no overlap
...
```

Code block 10: Fragment of our borrow-checker output for `get_or_insert.mir`. Note ... signifies omissions from output in code block 17.

Of note is the fact that we capture all expected borrows from `get_or_insert.rs`; the three explicit borrows at lines 2, 5, and 6 but interestingly also the indirect borrow of `v` from the MIR semantics interpretation of how the `HashMap`-lookup interacts with the match-pattern.

We see from the output of all *IN*- and *OUT*-equations for all `get_or_insert.mir` statements in 17 that the NLL-problematic borrow by location 2 of location 1 is only live in blocks $0, 1, 4$ as a consequence of the program trace of blocks $0 \rightarrow 1, \rightarrow 2 \rightarrow 4 \rightarrow 9$, where location 2 is used and thus killed by the phi-node of the Some-branch; block 4. This is the clear difference between NLL and our borrow-checker; we do not consider `v` from `get_or_insert.rs` live in the None-branch.

However, we do not consider location 2 live at the return as previously shown as the 'Goal-region' in Figure 2 from Section 4.1. We cannot reason about the lifetime of `v` after the return, at the caller, since we have not considered interprocedural borrow-checking analysis, just one function. We must at least consider whether we have a sound analysis by restricting the liveness of location 2 to, at most, the phi-node of the Some-case, and not at the return.

Yet, we obtain an borrow-checker which at first seems sound with no more complexity than the classic compiler liveness-analysis and some set theory in identifying and checking overlap of borrows. This fact is striking considering the lack of formalisms in defining both NLL and Polonius. We believe the field of borrow-checking is an interesting and under-explored domain within language formalisation and verification which warrants further investigation.

We have unfortunately merely argued, not proven the correctness of this analysis, which is required to properly underline and understand the results of this work.

We believe that the analysis could be correct depending on MIR semantics, for which a formal proof is delegated to future work.

7 Future Work

We will discuss facets of our contributions which we have identified could benefit with future work or require future work to be fully explored. These are

7.1 Soundness of analysis

Our borrow-checking analysis contribution would stand stronger by having a formal correctness proof, showing that analysis is indeed sound. In the negative case, if our analysis is not sound, we will have learned that this approach is not sufficient for capturing the borrowing rules of Rust.

7.2 Compare analysis against NLL test suite

We have not been able to compare our analysis against the same criteria that NLL or indeed Polonius are. This is due to the complexity involved in working with the Rust-compiler and extracting MIR for the programs in the suite.

In future, the results of such a test suite run would help indicate whether this analysis looses or gains accuracy compared to contemporary techniques. For Polonius, a repository of known problematic Rust programs are available that with mechanical translation into MIR could, with little effort, more thoroughly test `mir-owner-guillotine`.

7.3 Formalise Rusts borrowing system

Future work should contain effort into formalising the borrowing system of Rust, as seen in related work [23]. This would aid in both verifying the correctness of existing borrow-checkers such as NLL and Polonius but also assist in development of newer borrow-checkers.

7.4 Formalisation of MIR wrt. borrowing

In Section 3.4.1 we show that the simple, intuitive uses and definitions in MIR are sufficient for our analysis. However, for more complex statements a better and more rigorous approach would be welcome. Take the following statement s from `get_or_insert.mir`:

```
1 _2 = HashMap::<u32, String>::get::<u32>(move _3,
  ↪  move _5) -> bb1;
```

It's clear that $DEF(s) = \{2\}$, but later uses of _2 may or may not be used depending on their type. If the type, as shown here, is an integer it may be copied cheaply if multiple statements uses it. If it is a heap-stored value, it will require an explicit clone; `.clone()`, to satisfy multiple users without potential borrowing-issues.

A more rigorous understanding of MIR's semantics are required to properly argue the correctness and construct these DEF- and USE-functions.

7.5 Lattice liveness analysis

For liveness analysis as described in Section 3.4.1 we use dataflow-equations for all statements both before and after their execution as described in [1].

However, we could have taken the approach of lattice based liveness analysis as which we will show in the following. The construction of lattice and constraint rules for MIR syntax in this section are heavily inspired by toy imperative languages and constraint variables constructions, as described in [18].

We use the parameterised powerset lattice of locations which depends on the locations in the actual MIR program being analysed as shown in Equation 11 and Figure 6.

$$L = (\mathcal{P}(\{Locs\}), \subseteq) \tag{11}$$

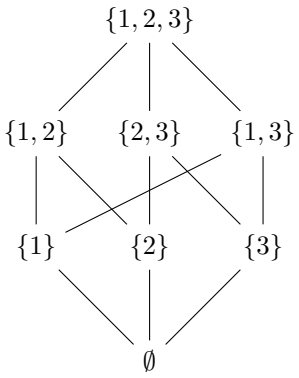


Figure 6: Example powerset lattice of locations for $Locs = \{1, 2, 3\}$

For any statement v in the nodes of our CFG we have a a constraint variable $[\![v]\!]$ that is the subset of $Locs$ which are live before this node.

We define $JOIN(v)$ in Equation 12 to handle the combining of live locations from successors to a node v in a merging branch. For this we define the order-relation as $\sqsubseteq = \subseteq$, which leaves us with $\sqcup = \cup$. [18]

$$JOIN(v) = \bigcup_{l \in succ(v)} [\![l]\!] \tag{12}$$

Additionally, we also need a utility function $uses(E)$ which denotes the locations used in an expression. This expression

We must define how the statements in MIR affect the liveness of locations. This is simple for the primitives as they only use locations and therefore generate liveness as seen in Equation 13.

$$\left.\begin{array}{l}\texttt{switchInt(E)];}\\ \texttt{drop(E);}\end{array}\right\} [\![l]\!] = JOIN(v) \cup uses(E) \tag{13}$$

Since MIR is mostly on SSA form, the bulk of statements are assignments. This The constraint rule for assignments is shown in Equation 14.

$$l_1 = E: \quad [\![v]\!] = JOIN(v) \setminus l_1 \cup uses(E) \tag{14}$$

The `return;` statement in MIR has an implicit use of location 0 and therefore we define its constraint rule as shown in Equation 15. We do not have any successors for a return as our analysis is intraprocedural, therefore we simply generate liveness for location 0.

$$\texttt{return;} \quad [\![v]\!] = 0 \tag{15}$$

For all other statements for which no locations are used nor any are defined, the constraint rule is simply as shown in Equation 16.

$$[\![v]\!] = JOIN(v) \tag{16}$$

We would need to show that our powerset lattice is a complete lattice, the functions $JOIN(v)$ and $locs(E)$ are monotonic, to apply the Kleene theorem [18] and know that our resulting least fixed point is unique.

8 Conclusion

In this work we have presented the ownership and borrowing concept of newer and well-liked [21] programming language; Rust. The main problem we aim to answer is; why is borrow-checking precision seemingly imprecise for unassuming code-patterns. The prime example is clearly `get_or_insert.rs`, also exemplified by rust-lang RFC2094 [15].

We make the the following three contributions in the effort to solve this problem:

8.1 Understanding NLL and Polonius

We believe the how and why NLL and Polonius works for borrow-checking is not entirely clear, which research into both verifying and formalising Rusts borrowing system support [23, 20, 5, 2].

In Sections 3.5 and 3.6 we explain as clearly as we are able, how these are defined, based on public documentation.

8.2 Liveness-based borrow-checking analysis

We define our own borrow-checking analysis based on liveness and tightly in contrast to definitions of NLL and Polonius, hence not based on established formalisms. We show how our analysis handles conditional branches and merges, loops, and how we identify borrowing errors.

8.3 Prototype of our borrow-checking analysis

We implement our liveness-based borrow-checking analysis into a proof-of-concept Python tool called `mir-owner-guillotine`. For parsing textual MIR, we define a grammar for textual MIR that, while restricted in the subset of MIR which it accepts, handles all statements that we have observed in our research textual output of MIR.

`mir-owner-guillotine` parses textual MIR input into our own IR, which lends convenient structures and functions for liveness and our borrow-checking analysis. If not resource limited, we would have liked for `mir-owner-guillotine` to be integrated into the Rust compiler.

Acknowledgements

We would like to thank our supervisors René Rydhof Hansen and Danny Bøgsted Poulsen for their guidance and feedback.

References

[1] Andrew W Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[2] Vytautas Astrauskas et al. 'Leveraging Rust types for modular specification and verification'. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–30.

[3] Cristian Cadar, Daniel Dunbar, Dawson R Engler et al. 'Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.' In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[4] Dave Clarke and Sophia Drossopoulou. 'Ownership, encapsulation and the disjointness of type and effect'. In: *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2002, pp. 292–310.

[5] Simon Vinberg Andersen Felix Cho Petersen Mathias Knøsgaard Kristensen. 'Rust's Borrow System in Static Analysis'. PhD thesis. Master's thesis. Aalborg University, 2022.

[6] integer32llc. *rust-playground*. `https://github.com/integer32llc/rust-playground`. [Online; accessed 09-01-2023]. 2023.

[7] rust lang. *Ownership and moves*. `https://doc.rust-lang.org/rust-by-example/scope/move.html`. [Online; accessed 19-01-2023]. 2023.

[8] rust lang. *The Polonius Book*. `https://rust-lang.github.io/polonius/`. [Online; accessed 18-01-2023]. 2023.

[9] rust lang. *The Rust Book: Understanding Ownership*. `https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html`. [Online; accessed 19-01-2023]. 2023.

[10] rust lang/polonius. *polonius/inputs/issue-47680/issue-47680.rs*. `https://github.com/rust-lang/polonius/blob/master/inputs/issue-47680/issue-47680.rs`. [Online; accessed 19-01-2023]. 2023.

[11] Zhuohua Li et al. 'MirChecker: detecting bugs in Rust programs via static analysis'. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 2183–2196.

[12] Marcus Lindner, Jorge Aparicius and Per Lindgren. 'No panic! Verification of Rust programs by symbolic execution'. In: *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. IEEE. 2018, pp. 108–114.

[13] Nico Matsakis. *An alias-based formulation of the borrow checker*. `http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/`. [Online; accessed 20-12-2022]. 2018.

[14] Nico Matsakis. *RFC 1211*. `https://rust-lang.github.io/rfcs/1211-mir.html`. [Online; accessed 19-12-2022]. 2015.

[15] Nico Matsakis. *RFC 2094*. `https://rust-lang.github.io/rfcs/2094-nll.html`. [Online; accessed 20-12-2022]. 2017.

[16] Niko Matsakis. *Non-lexical lifetimes (NLL) fully stable*. `https://blog.rust-lang.org/2022/08/05/nll-by-default.html`. [Online; accessed 19-12-2022]. 2022.

[17] Niko Matsakis. *Rust Belt Rust Conference Talk: Polonius: Either Borrower or Lender Be, but Responsibly*. Youtube. URL: `https://www.youtube.com/watch?v=_agDeiWek8w&t`.

[18] Anders Møller and Michael I Schwartzbach. 'Static program analysis'. In: *Notes. Feb* (2012).

[19] Peter Müller, Malte Schwerhoff and Alexander J Summers. 'Viper: A verification infrastructure for permission-based reasoning'. In: *International conference on verification, model checking, and abstract interpretation*. Springer. 2016, pp. 41–62.

[20] Emil Jørgensen Njor and Hilmar Gústafsson. 'Static Taint Analysis in Rust'. PhD thesis. Master's thesis. Aalborg University, 2021.

[21] Stack Overflow. *Most loved, dreaded, and wanted*. `https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages`. [Online; accessed 19-12-2022]. 2022.

[22] Matthew J Parkinson and Alexander J Summers. 'The relationship between separation logic and implicit dynamic frames'. In: *European Symposium on Programming*. Springer. 2011, pp. 439–458.

[23] David J Pearce. 'A lightweight formalism for reference lifetimes and borrowing in Rust'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43.1 (2021), pp. 1–73.

[24] Henry Gordon Rice. 'Classes of recursively enumerable sets and their decision problems'. In: *Transactions of the American Mathematical society* 74.2 (1953), pp. 358–366.

[25] Jan Smans, Bart Jacobs and Frank Piessens. 'Implicit dynamic frames: Combining dynamic frames and separation logic'. In: *European Conference on Object-Oriented Programming*. Springer. 2009, pp. 148–172.

A MIR grammar

```
<mir>       ::= <function>
<function>  ::= "fn" <name> "(" <param> ")" "->" <ret-val> "{" <bblist> "}"

<param>     ::= (<name> ("," <name>)*)+
<stmtlist>  ::= <stmtlist> <statement> | <statement>
<bblist>    ::= <bblist> <block> | <block>
<statement> ::= <LOCATION> "=" <stmttype> ; | "goto" "->" <bb> ";" | "unreachable" ";" | "return" |
  ↪  "primitives" | "assert" ;

<stmttype>  ::= <LOCATION> | <constant> | <borrow> | unreachable | return | function_call | move

constant    ::= CONST NUMBER _ TYPE
borrow      ::= REF source | REFMUT source
source      ::= ( source ) | LOCATION | DEREF LOCATION


move        ::= MOVE "(" valueargs ")"

function_call ::= generic COLONTWICE method_call ( valueargs ) goto_block
method_call ::= METHODNAME | METHODNAME turbofish
turbofish   ::= COLONTWICE "<" typeargs ">"
generic     ::= generic COLONTWICE "<" typeargs ">" cast
                | TYPENAMES "<" typeargs ">" cast
                | TYPENAMES
                | generic cast
                | "<" generic ">"
cast        ::= AS TYPENAMES < typeargs > | empty
typeargs    ::= typearg "," typeargs | typearg
typearg     ::= TYPENAMES | REF TYPENAMES | REFMUT TYPENAMES

method_call ::= METHODNAME | METHODNAME turbofish
turbofish   ::= COLONTWICE "<" typeargs ">"
typeargs    ::= typearg "," typeargs | typearg
typearg     ::= TYPENAMES | REF TYPENAMES | REFMUT TYPENAMES
valueargs   ::= valueargs "," valuearg | valuearg
valuearg    ::= LOCATION | MOVE LOCATION | valuearg_constant (reuse from stmt)
             | CONST STRING | STRING
             | mode "(" LOCATION "." NUMBER ":" TYPENAMES ")"
             | mode "(" LOCATION "." NUMBER ":" TYPENAMES ")"
goto_block  ::= ARROW BB
goto_cond_block ::= ARROW "[" goto_params "]"
goto_params ::= goto_params "," goto_param | goto_param

goto_param  ::= NUMBER "_" TYPENAMES ":" BB | OTHERWISE ":" BB
```

Code block 11: MIR EBNF created for `mir-owner-guillotine`

B Conceptual UML diagram `mir-owner-guillotine` IR

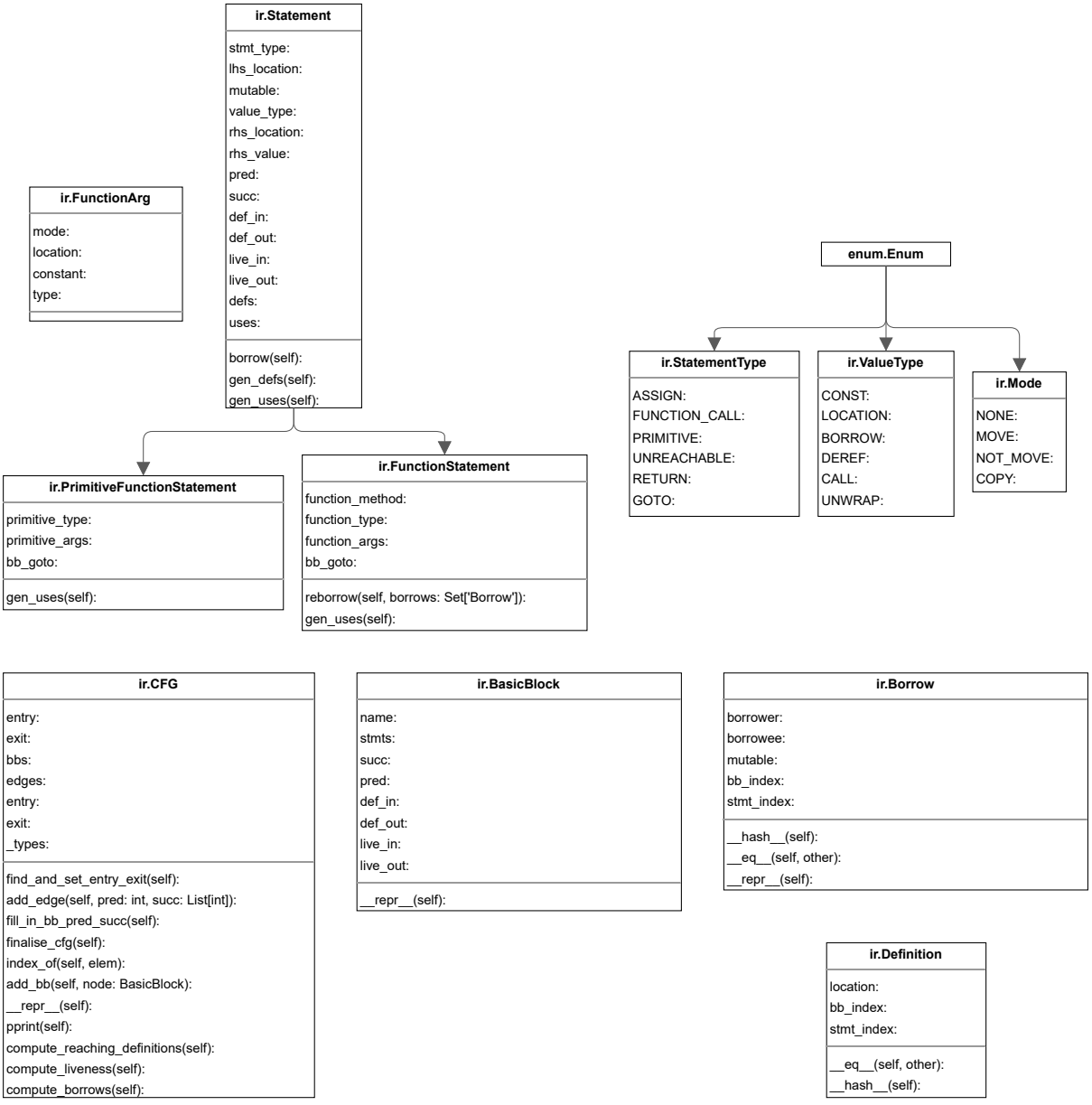


Figure 7: UML diagram of `mir-owner-guillotine` IR

C `get_or_insert.mir` program construction

To prepare `get_or_insert.mir` for analysis by `mir-owner-guillotine` we have to modify it, since NLL rejects the program.

We construct to imagined Rust programs from `get_or_insert.rs`; MIR-A in code block 12 and MIR-B in code block 13. For completion, we add a required main function and import the HashMap standard library type that we use for `get_or_insert.rs`.

```
1 use std::collections::HashMap;
2 fn main() {}
3 fn get_or_insert (map: &mut HashMap<u32, String>) -> &String {
4  match HashMap::get(&*map, &22) {
5   Some(v) => v,
6   None => {
7    //map.insert(42, String::from("init"));
8    &map[&42]
9   }
10  }
11 }
```

Code block 12: MIR-A Rust program, map insert at line 5 commented out

```
1 use std::collections::HashMap;
2 fn main() {}
3 fn get_or_insert (map: &mut HashMap<u32, String>) -> &String {
4  match HashMap::get(&*map, &22) {
5   Some(v) => &map[&42],
6   None => {
7    map.insert(42, String::from("init"));
8    &map[&42]
9   }
10  }
11 }
```

Code block 13: MIR-B Rust program. Return of control-flow reference reborrowed into `v` at line3 is removed. Instead we remove a fresh reference of the same location in the map: `&map[&42]`.

Below in code blocks 14 and 15, we have the full - and unfortunately very verbose - output of the MIR textual generation from rust-playground. We believe it is important to show the entirety of the generated MIR for completeness.

```
1 fn main() -> () {
2     let mut _0: ();                      // return place in scope 0 at src/main.rs:2:11: 2:11
3
4     bb0: {
5         return;                          // scope 0 at src/main.rs:2:13: 2:13
6     }
7 }
8
9 fn get_or_insert(_1: &mut HashMap<u32, String>) -> &String {
10     debug map => _1;                     // in scope 0 at src/main.rs:3:19: 3:22
11     let mut _0: &std::string::String;    // return place in scope 0 at src/main.rs:3:55: 3:63
12     let mut _2: std::option::Option<&std::string::String>; // in scope 0 at src/main.rs:5:9: 5:33
```

```
13      let mut _3: &std::collections::HashMap<u32, std::string::String>; // in scope 0 at src/main.rs:5:22:
        ↪  5:27
14      let _4: &std::collections::HashMap<u32, std::string::String>; // in scope 0 at src/main.rs:5:22: 5:27
15      let mut _5: &u32;                     // in scope 0 at src/main.rs:5:29: 5:32
16      let _6: &u32;                         // in scope 0 at src/main.rs:5:29: 5:32
17      let mut _7: isize;                    // in scope 0 at src/main.rs:6:5: 6:12
18      let _8: &std::string::String;         // in scope 0 at src/main.rs:6:10: 6:11
19      let _9: &std::string::String;         // in scope 0 at src/main.rs:9:7: 9:16
20      let _10: &std::string::String;        // in scope 0 at src/main.rs:9:8: 9:16
21      let mut _11: &std::collections::HashMap<u32, std::string::String>; // in scope 0 at src/main.rs:9:8:
        ↪  9:11
22      let mut _12: &u32;                    // in scope 0 at src/main.rs:9:12: 9:15
23      let _13: &u32;                        // in scope 0 at src/main.rs:9:12: 9:15
24      let mut _14: &u32;                    // in scope 0 at src/main.rs:9:12: 9:15
25      let mut _15: &u32;                    // in scope 0 at src/main.rs:5:29: 5:32
26      scope 1 {
27          debug v => _8;                    // in scope 1 at src/main.rs:6:10: 6:11
28      }
29
30      bb0: {
31          _4 = &(*_1);                      // scope 0 at src/main.rs:5:22: 5:27
32          _3 = _4;                          // scope 0 at src/main.rs:5:22: 5:27
33          _15 = const _;                    // scope 0 at src/main.rs:5:29: 5:32
34                                            // mir::Constant
35                                            // + span: src/main.rs:5:29: 5:32
36                                            // + literal: Const { ty: &u32, val: Unevaluated(get_or_insert,
                                              ↪  [], Some(promoted[1])) }
37          _6 = _15;                         // scope 0 at src/main.rs:5:29: 5:32
38          _5 = _6;                          // scope 0 at src/main.rs:5:29: 5:32
39          _2 = HashMap::<u32, String>::get::<u32>(move _3, move _5) -> bb1; // scope 0 at src/main.rs:5:9:
            ↪  5:33
40                                            // mir::Constant
41                                            // + span: src/main.rs:5:9: 5:21
42                                            // + user_ty: UserType(0)
43                                            // + literal: Const { ty: for<'a, 'b> fn(&'a HashMap<u32,
                                              ↪  String>, &'b u32) -> Option<&'a String> {HashMap::<u32,
                                              ↪  String>::get::<u32>}, val: Value(<ZST>) }
44      }
45
46      bb1: {
47          _7 = discriminant(_2);            // scope 0 at src/main.rs:5:9: 5:33
48          switchInt(move _7) -> [0_isize: bb2, 1_isize: bb4, otherwise: bb3]; // scope 0 at
            ↪  src/main.rs:5:3: 5:33
49      }
50
51      bb2: {
52          _11 = &(*_1);                     // scope 0 at src/main.rs:9:8: 9:11
53          _14 = const _;                    // scope 0 at src/main.rs:9:12: 9:15
54                                            // mir::Constant
55                                            // + span: src/main.rs:9:12: 9:15
56                                            // + literal: Const { ty: &u32, val: Unevaluated(get_or_insert,
                                              ↪  [], Some(promoted[0])) }
57          _13 = _14;                        // scope 0 at src/main.rs:9:12: 9:15
58          _12 = _13;                        // scope 0 at src/main.rs:9:12: 9:15
59          _10 = <HashMap<u32, String> as Index<&u32>>::index(move _11, move _12) -> bb5; // scope 0 at
            ↪  src/main.rs:9:8: 9:16
60                                            // mir::Constant
61                                            // + span: src/main.rs:9:8: 9:16
```

```
                                        // + literal: Const { ty: for<'a> fn(&'a HashMap<u32, String>,
                                        ↪  &u32) -> &'a <HashMap<u32, String> as Index<&u32>>::Output
                                        ↪  {<HashMap<u32, String> as Index<&u32>>::index}, val:
                                        ↪  Value(<ZST>) }
    }

    bb3: {
        unreachable;                    // scope 0 at src/main.rs:5:9: 5:33
    }

    bb4: {
        _8 = ((_2 as Some).0: &std::string::String); // scope 0 at src/main.rs:6:10: 6:11
        _0 = _8;                        // scope 1 at src/main.rs:6:16: 6:17
        goto -> bb6;                    // scope 0 at src/main.rs:6:16: 6:17
    }

    bb5: {
        _9 = _10;                       // scope 0 at src/main.rs:9:7: 9:16
        _0 = _9;                        // scope 0 at src/main.rs:9:7: 9:16
        goto -> bb6;                    // scope 0 at src/main.rs:10:5: 10:6
    }

    bb6: {
        return;                         // scope 0 at src/main.rs:12:2: 12:2
    }
}

promoted[0] in get_or_insert: &u32 = {
    let mut _0: &u32;                   // return place in scope 0 at src/main.rs:9:12: 9:15
    let mut _1: u32;                    // in scope 0 at src/main.rs:9:13: 9:15

    bb0: {
        _1 = const 42_u32;              // scope 0 at src/main.rs:9:13: 9:15
        _0 = &_1;                       // scope 0 at src/main.rs:9:12: 9:15
        return;                         // scope 0 at src/main.rs:9:12: 9:15
    }
}

promoted[1] in get_or_insert: &u32 = {
    let mut _0: &u32;                   // return place in scope 0 at src/main.rs:5:29: 5:32
    let mut _1: u32;                    // in scope 0 at src/main.rs:5:30: 5:32

    bb0: {
        _1 = const 22_u32;              // scope 0 at src/main.rs:5:30: 5:32
        _0 = &_1;                       // scope 0 at src/main.rs:5:29: 5:32
        return;                         // scope 0 at src/main.rs:5:29: 5:32
    }
}
```

Code block 14: MIR-A MIR output from rust-playground

```
fn main() -> () {
    let mut _0: ();                      // return place in scope 0 at src/main.rs:2:11: 2:11

    bb0: {
        return;                          // scope 0 at src/main.rs:2:13: 2:13
    }
}

fn get_or_insert(_1: &mut HashMap<u32, String>) -> &String {
    debug map => _1;                     // in scope 0 at src/main.rs:3:19: 3:22
    let mut _0: &std::string::String;    // return place in scope 0 at src/main.rs:3:55: 3:63
    let mut _2: std::option::Option<&std::string::String>; // in scope 0 at src/main.rs:5:9: 5:33
    let mut _3: &std::collections::HashMap<u32, std::string::String>; // in scope 0 at src/main.rs:5:22:
    ↪  5:27
    let _4: &std::collections::HashMap<u32, std::string::String>; // in scope 0 at src/main.rs:5:22: 5:27
    let mut _5: &u32;                    // in scope 0 at src/main.rs:5:29: 5:32
    let _6: &u32;                        // in scope 0 at src/main.rs:5:29: 5:32
    let mut _7: isize;                   // in scope 0 at src/main.rs:6:5: 6:12
    let _8: &std::string::String;        // in scope 0 at src/main.rs:6:10: 6:11
    let _9: &std::string::String;        // in scope 0 at src/main.rs:6:16: 6:25
    let _10: &std::string::String;       // in scope 0 at src/main.rs:6:17: 6:25
    let mut _11: &std::collections::HashMap<u32, std::string::String>; // in scope 0 at src/main.rs:6:17:
    ↪  6:20
    let mut _12: &u32;                   // in scope 0 at src/main.rs:6:21: 6:24
    let _13: &u32;                       // in scope 0 at src/main.rs:6:21: 6:24
    let _14: std::option::Option<std::string::String>; // in scope 0 at src/main.rs:8:7: 8:43
    let mut _15: &mut std::collections::HashMap<u32, std::string::String>; // in scope 0 at
    ↪  src/main.rs:8:7: 8:43
    let mut _16: std::string::String;    // in scope 0 at src/main.rs:8:22: 8:42
    let _17: &std::string::String;       // in scope 0 at src/main.rs:9:7: 9:16
    let _18: &std::string::String;       // in scope 0 at src/main.rs:9:8: 9:16
    let mut _19: &std::collections::HashMap<u32, std::string::String>; // in scope 0 at src/main.rs:9:8:
    ↪  9:11
    let mut _20: &u32;                   // in scope 0 at src/main.rs:9:12: 9:15
    let _21: &u32;                       // in scope 0 at src/main.rs:9:12: 9:15
    let mut _22: &u32;                   // in scope 0 at src/main.rs:9:12: 9:15
    let mut _24: &u32;                   // in scope 0 at src/main.rs:5:29: 5:32
    scope 1 {
        debug v => _8;                   // in scope 1 at src/main.rs:6:10: 6:11
        let mut _23: &u32;               // in scope 1 at src/main.rs:6:21: 6:24
    }

    bb0: {
        _4 = &(*_1);                     // scope 0 at src/main.rs:5:22: 5:27
        _3 = _4;                         // scope 0 at src/main.rs:5:22: 5:27
        _24 = const _;                   // scope 0 at src/main.rs:5:29: 5:32
                                         // mir::Constant
                                         // + span: src/main.rs:5:29: 5:32
                                         // + literal: Const { ty: &u32, val: Unevaluated(get_or_insert,
                                         ↪  [], Some(promoted[2])) }
        _6 = _24;                        // scope 0 at src/main.rs:5:29: 5:32
        _5 = _6;                         // scope 0 at src/main.rs:5:29: 5:32
        _2 = HashMap::<u32, String>::get::<u32>(move _3, move _5) -> bb1; // scope 0 at src/main.rs:5:9:
        ↪  5:33
                                         // mir::Constant
                                         // + span: src/main.rs:5:9: 5:21
```

```
51                                      // + user_ty: UserType(0)
52                                      // + literal: Const { ty: for<'a, 'b> fn(&'a HashMap<u32,
                                        ↪  String>, &'b u32) -> Option<&'a String> {HashMap::<u32,
                                        ↪  String>::get::<u32>}, val: Value(<ZST>) }
53      }
54
55      bb1: {
56          _7 = discriminant(_2);          // scope 0 at src/main.rs:5:9: 5:33
57          switchInt(move _7) -> [0_isize: bb2, 1_isize: bb4, otherwise: bb3]; // scope 0 at
            ↪  src/main.rs:5:3: 5:33
58      }
59
60      bb2: {
61          _15 = &mut (*_1);               // scope 0 at src/main.rs:8:7: 8:43
62          _16 = <String as From<&str>>::from(const "init") -> bb6; // scope 0 at src/main.rs:8:22: 8:42
63                                          // mir::Constant
64                                          // + span: src/main.rs:8:22: 8:34
65                                          // + user_ty: UserType(1)
66                                          // + literal: Const { ty: fn(&str) -> String {<String as
                                            ↪  From<&str>>::from}, val: Value(<ZST>) }
67                                          // mir::Constant
68                                          // + span: src/main.rs:8:35: 8:41
69                                          // + literal: Const { ty: &str, val: Value(Slice(..)) }
70      }
71
72      bb3: {
73          unreachable;                    // scope 0 at src/main.rs:5:9: 5:33
74      }
75
76      bb4: {
77          _8 = ((_2 as Some).0: &std::string::String); // scope 0 at src/main.rs:6:10: 6:11
78          _11 = &(*_1);                   // scope 1 at src/main.rs:6:17: 6:20
79          _23 = const _;                  // scope 1 at src/main.rs:6:21: 6:24
80                                          // mir::Constant
81                                          // + span: src/main.rs:6:21: 6:24
82                                          // + literal: Const { ty: &u32, val: Unevaluated(get_or_insert,
                                            ↪  [], Some(promoted[1])) }
83          _13 = _23;                      // scope 1 at src/main.rs:6:21: 6:24
84          _12 = _13;                      // scope 1 at src/main.rs:6:21: 6:24
85          _10 = <HashMap<u32, String> as Index<&u32>>::index(move _11, move _12) -> bb5; // scope 1 at
            ↪  src/main.rs:6:17: 6:25
86                                          // mir::Constant
87                                          // + span: src/main.rs:6:17: 6:25
88                                          // + literal: Const { ty: for<'a> fn(&'a HashMap<u32, String>,
                                            ↪  &u32) -> &'a <HashMap<u32, String> as Index<&u32>>::Output
                                            ↪  {<HashMap<u32, String> as Index<&u32>>::index}, val:
                                            ↪  Value(<ZST>) }
89      }
90
91      bb5: {
92          _9 = _10;                       // scope 1 at src/main.rs:6:16: 6:25
93          _0 = _9;                        // scope 1 at src/main.rs:6:16: 6:25
94          goto -> bb10;                   // scope 0 at src/main.rs:6:24: 6:25
95      }
96
97      bb6: {
98          _14 = HashMap::<u32, String>::insert(move _15, const 42_u32, move _16) -> bb7; // scope 0 at
            ↪  src/main.rs:8:7: 8:43
```

```
99                                          // mir::Constant
100                                         // + span: src/main.rs:8:11: 8:17
101                                         // + literal: Const { ty: for<'a> fn(&'a mut HashMap<u32,
                                            ↪  String>, u32, String) -> Option<String> {HashMap::<u32,
                                            ↪  String>::insert}, val: Value(<ZST>) }
102     }
103
104     bb7: {
105         drop(_14) -> bb8;               // scope 0 at src/main.rs:8:43: 8:44
106     }
107
108     bb8: {
109         _19 = &(*_1);                   // scope 0 at src/main.rs:9:8: 9:11
110         _22 = const _;                  // scope 0 at src/main.rs:9:12: 9:15
111                                         // mir::Constant
112                                         // + span: src/main.rs:9:12: 9:15
113                                         // + literal: Const { ty: &u32, val: Unevaluated(get_or_insert,
                                            ↪  [], Some(promoted[0])) }
114         _21 = _22;                      // scope 0 at src/main.rs:9:12: 9:15
115         _20 = _21;                      // scope 0 at src/main.rs:9:12: 9:15
116         _18 = <HashMap<u32, String> as Index<&u32>>::index(move _19, move _20) -> bb9; // scope 0 at
            ↪  src/main.rs:9:8: 9:16
117                                         // mir::Constant
118                                         // + span: src/main.rs:9:8: 9:16
119                                         // + literal: Const { ty: for<'a> fn(&'a HashMap<u32, String>,
                                            ↪  &u32) -> &'a <HashMap<u32, String> as Index<&u32>>::Output
                                            ↪  {<HashMap<u32, String> as Index<&u32>>::index}, val:
                                            ↪  Value(<ZST>) }
120     }
121
122     bb9: {
123         _17 = _18;                      // scope 0 at src/main.rs:9:7: 9:16
124         _0 = _17;                       // scope 0 at src/main.rs:9:7: 9:16
125         goto -> bb10;                   // scope 0 at src/main.rs:10:5: 10:6
126     }
127
128     bb10: {
129         return;                         // scope 0 at src/main.rs:12:2: 12:2
130     }
131 }
132
133 promoted[0] in get_or_insert: &u32 = {
134     let mut _0: &u32;                   // return place in scope 0 at src/main.rs:9:12: 9:15
135     let mut _1: u32;                    // in scope 0 at src/main.rs:9:13: 9:15
136
137     bb0: {
138         _1 = const 42_u32;              // scope 0 at src/main.rs:9:13: 9:15
139         _0 = &_1;                       // scope 0 at src/main.rs:9:12: 9:15
140         return;                         // scope 0 at src/main.rs:9:12: 9:15
141     }
142 }
143
144 promoted[1] in get_or_insert: &u32 = {
145     let mut _0: &u32;                   // return place in scope 0 at src/main.rs:6:21: 6:24
146     let mut _1: u32;                    // in scope 0 at src/main.rs:6:22: 6:24
147
148     bb0: {
149         _1 = const 42_u32;              // scope 0 at src/main.rs:6:22: 6:24
```

```
150         _0 = &_1;                        // scope 0 at src/main.rs:6:21: 6:24
151         return;                          // scope 0 at src/main.rs:6:21: 6:24
152     }
153 }
154
155 promoted[2] in get_or_insert: &u32 = {
156     let mut _0: &u32;                    // return place in scope 0 at src/main.rs:5:29: 5:32
157     let mut _1: u32;                     // in scope 0 at src/main.rs:5:30: 5:32
158
159     bb0: {
160         _1 = const 22_u32;               // scope 0 at src/main.rs:5:30: 5:32
161         _0 = &_1;                        // scope 0 at src/main.rs:5:29: 5:32
162         return;                          // scope 0 at src/main.rs:5:29: 5:32
163     }
164 }
```

Code block 15: MIR-B MIR output from rust-playground

Applying our hand-crafted MIR construction from Section 4.2, we complete the following modifications to MIR-A to render our final MIR for `get_or_insert.mir`:

We extract the blocks from MIR-B denoting the insertion of the default value. This is blocks 2, 6, 7. We see that the blocks 8 and 9 in MIR-B are identical to blocks 2 and 5 in MIR-A, with the exception of location numbering. These two pairs of blocks are resulting from the `&map[&42]` returning statement in line 8 for both code block 12 and 13. We also see the use of promoted constants for `&42` which we simply inline into the HashMap insert in line 29 in code block 16.

Next we remove the type declarations and insert *dummy* line 2 in code block 16 to emulate the map argument. This is required because we do not support type declarations in `mir-owner-guillotine`.

Lastly we insert the extracted blocks from MIR-B identified above, that is blocks 2, 6, and 7 which we insert at as-is, the successor in the None-case of the switchInt in 1, line 10 in code block 16. We also insert 8 and 9 from MIR-B for the `&map[&42]` returning statement.

This leaves us with MIR-A, with blocks 2 and 5 removed, with blocks 2, 6, 7, 8, 9 from MIR-B inserted and thus we need to adjust block numbering to account for the insertion of these five blocks. The identified subgraph from MIR-B is transformed: $2, 6 \rightarrow 5, 7, 8, 9$. The entirety of the MIR-A graph sans the `&map[&42]` returning statement is slightly transformed: blocks $0, 1, 3, 4, 6 \rightarrow 9$. For both transformations we change the index, predecessor- and successor control-flow primitives to reflect this change.

Finally, we are rendered the resulting MIR in code block 16.

```
bb0: {
    _1 = HashMap::new(const 42_u32);
    _3 = &(*_1);
    _5 = const 42_u32;
    _2 = HashMap::<u32, String>::get::<u32>(move _3, move _5) -> bb1;
}

bb1: {
    _7 = discriminant(_2);
    switchInt(move _7) -> [0_isize: bb2, 1_isize: bb4, otherwise: bb3];
}

bb2: {
    _17 = &mut (*_1);
    _18 = <String as From<&str>>::from(const "init") -> bb5;
}

bb3: {
    unreachable;
}

bb4: {
    _8 = ((_2 as Some).0: &std::string::String);
    _0 = _8;
    goto -> bb9;
}

bb5: {
    _16 = HashMap::<u32, String>::insert(move _17, const 42_u32, move _18) -> bb6;
}

bb6: {
    drop(_16) -> bb7;
}

bb7: {
    _11 = &(*_1);
    _12 = const 42_u32;
    _10 = <HashMap<u32, String> as Index<&u32>>::index(move _11, move _12) -> bb8;
}

bb8: {
    _0 = _10;
    goto -> bb9;
}

bb9: {
    return;
}
```

Code block 16: `get_or_insert.mir`

```
bb 0 liveness:
        stmt:0 live in: set()          live out: {1}
        stmt:1 live in: {1}          live out: {1, 3}
        stmt:2 live in: {1, 3}          live out: {1, 3, 5}
        stmt:3 live in: {1, 3, 5}          live out: {1, 2}
==========
bb 1 liveness:
        stmt:0 live in: {1, 2}          live out: {1, 2, 7}
        stmt:1 live in: {1, 2, 7}          live out: {1, 2}
==========
bb 2 liveness:
        stmt:0 live in: {1}          live out: {17, 1}
        stmt:1 live in: {17, 1}          live out: {17, 18, 1}
==========
bb 3 liveness:
        stmt:0 live in: set()          live out: set()
==========
bb 4 liveness:
        stmt:0 live in: {2}          live out: {8}
        stmt:1 live in: {8}          live out: {0}
        stmt:2 live in: {0}          live out: {0}
==========
bb 5 liveness:
        stmt:0 live in: {17, 18, 1}          live out: {16, 1}
==========
bb 6 liveness:
        stmt:0 live in: {16, 1}          live out: {1}
==========
bb 7 liveness:
        stmt:0 live in: {1}          live out: {11}
        stmt:1 live in: {11}          live out: {11, 12}
        stmt:2 live in: {11, 12}          live out: {10}
==========
bb 8 liveness:
        stmt:0 live in: {10}          live out: {0}
        stmt:1 live in: {0}          live out: {0}
==========
bb 9 liveness:
        stmt:0 live in: {0}          live out: set()
==========
Borrow-checking:
Bcking borrowee 1 with borrows:
        Borrow(er=3, ee=1, mut=False, bb0, s1)
        Borrow(er=2, ee=1, mut=False, bb0, s3)
        Borrow(er=11, ee=1, mut=False, bb7, s0)
        Borrow(er=17, ee=1, mut=True, bb2, s0)
checking         Borrow(er=3, ee=1, mut=False, bb0, s1)
and                 Borrow(er=2, ee=1, mut=False, bb0, s3)
both immutable, don't care about overlap

checking         Borrow(er=3, ee=1, mut=False, bb0, s1)
and                 Borrow(er=11, ee=1, mut=False, bb7, s0)
both immutable, don't care about overlap

checking         Borrow(er=3, ee=1, mut=False, bb0, s1)
and                 Borrow(er=17, ee=1, mut=True, bb2, s0)
```

```
57 no overlap
58
59 checking         Borrow(er=2, ee=1, mut=False, bb0, s3)
60 and                 Borrow(er=11, ee=1, mut=False, bb7, s0)
61 both immutable, don't care about overlap
62
63 checking         Borrow(er=2, ee=1, mut=False, bb0, s3)
64 and                 Borrow(er=17, ee=1, mut=True, bb2, s0)
65 no overlap
66
67 checking         Borrow(er=11, ee=1, mut=False, bb7, s0)
68 and                 Borrow(er=17, ee=1, mut=True, bb2, s0)
69 no overlap
70
71 Bcking borrowee 2 with borrows:
72         Borrow(er=8, ee=2, mut=False, bb4, s0)
73 no mutable borrows for: 2, have 1 immutable borrows, all good.
74 borrow-check thinks program is valid? True
```

Code block 17: Result of `mir-owner-guillotine` by passing `get_or_insert.mir` as input

D Loop-cond-mut (issue-47680) program construction

For development of Polonius, a suite of test-cases are constructed. The following example is known as Issue-47680[2].

NLL rejects the program, despite being able to handle the conditional mutation of a control-flow variable in the simple if-statement case. However, when looping the same construct, NLL rejects it. This limitation is also the case for the `get_or_insert.mir`-program.

```
1  struct Thing;
2
3  impl Thing {
4      fn maybe_next(&mut self) -> Option<&mut Self> { None }
5  }
6
7  fn main() {
8      let mut temp = &mut Thing;
9
10     loop {
11         match temp.maybe_next() {
12             Some(v) => { temp = v; }
13             None => { }
14         }
15     }
16 }
```

Code block 18: Rust source code for loop with conditional mutation

Above Rust does not compiler under NLL, so to obtain MIR for analysis we simply introduce a mutable variable `a` and mutate that instead of the `temp` variable which is used in the conditional control flow at line 11. The resulting MIR-A is shown in code block 19

```
1  struct Thing;
2
3  impl Thing {
4      fn maybe_next(&mut self) -> Option<&mut Self> { None }
5  }
6
7  fn main() {
8      let mut temp = &mut Thing;
9      let mut a = 0;
10
11     loop {
12         match temp.maybe_next() {
13             Some(v) => { a = 1; }
14             None => { }
15         }
16     }
17 }
```

Code block 19: Modified loop-cond-mut example with substituting mutation onto other variable

[2] https://github.com/rust-lang/polonius/blob/master/inputs/issue-47680/issue-47680.rs

```
fn <impl at src/main.rs:3:1: 3:11>::maybe_next(_1: &mut Thing) -> Option<&mut Thing> {
    debug self => _1;                    // in scope 0 at src/main.rs:4:19: 4:28
    let mut _0: std::option::Option<&mut Thing>; // return place in scope 0 at src/main.rs:4:33: 4:50

    bb0: {
        Deinit(_0);                      // scope 0 at src/main.rs:4:53: 4:57
        discriminant(_0) = 0;            // scope 0 at src/main.rs:4:53: 4:57
        return;                          // scope 0 at src/main.rs:4:59: 4:59
    }
}

fn main() -> () {
    let mut _0: ();                      // return place in scope 0 at src/main.rs:7:11: 7:11
    let mut _1: &mut Thing;              // in scope 0 at src/main.rs:8:9: 8:17
    let mut _2: Thing;                   // in scope 0 at src/main.rs:8:25: 8:30
    let mut _4: std::option::Option<&mut Thing>; // in scope 0 at src/main.rs:12:11: 12:28
    let mut _5: &mut Thing;              // in scope 0 at src/main.rs:12:11: 12:28
    let mut _6: isize;                   // in scope 0 at src/main.rs:13:9: 13:16
    scope 1 {
        debug temp => _1;                // in scope 1 at src/main.rs:8:9: 8:17
        let mut _3: i32;                 // in scope 1 at src/main.rs:9:9: 9:14
        scope 2 {
            debug a => _3;               // in scope 2 at src/main.rs:9:9: 9:14
            let _7: &mut Thing;          // in scope 2 at src/main.rs:13:14: 13:15
            scope 3 {
                debug v => _7;           // in scope 3 at src/main.rs:13:14: 13:15
            }
        }
    }

    bb0: {
        _1 = &mut _2;                    // scope 0 at src/main.rs:8:20: 8:30
        _3 = const 0_i32;                // scope 1 at src/main.rs:9:17: 9:18
        goto -> bb1;                     // scope 2 at src/main.rs:11:5: 16:6
    }

    bb1: {
        _5 = &mut (*_1);                 // scope 2 at src/main.rs:12:11: 12:28
        _4 = Thing::maybe_next(move _5) -> bb2; // scope 2 at src/main.rs:12:11: 12:28
                                         // mir::Constant
                                         // + span: src/main.rs:12:16: 12:26
                                         // + literal: Const { ty: for<'a> fn(&'a mut Thing) ->
                                         ↪  Option<&'a mut Thing> {Thing::maybe_next}, val: Value(<ZST>)
                                         ↪  }
    }

    bb2: {
        _6 = discriminant(_4);           // scope 2 at src/main.rs:12:11: 12:28
        switchInt(move _6) -> [0_isize: bb1, 1_isize: bb4, otherwise: bb3]; // scope 2 at
        ↪  src/main.rs:12:5: 12:28
    }

    bb3: {
        unreachable;                     // scope 2 at src/main.rs:12:11: 12:28
    }

```

```
54      bb4: {
55          _7 = move ((_4 as Some).0: &mut Thing); // scope 2 at src/main.rs:13:14: 13:15
56          _3 = const 1_i32;                      // scope 3 at src/main.rs:13:22: 13:27
57          goto -> bb1;                           // scope 2 at src/main.rs:13:29: 13:30
58      }
59  }
```

Code block 20: Resulting MIR of modified loop-cond-mut program

Unfortunately, `mir-owner-guillotine` is not mature enough to handle all the constructs of the MIR. We therefore omit function signatures and location type-declarations s.t. we render only basic blocks. We can only analyse a single function at a time, so the removal of `maybe_next`-function on the `Thing` struct is required.

We must also modify the MIR to mutate the `temp` variable instead of our inserted `a` in the resulting MIR in code block 20 of modified program in code block 19. This is achieved by removing all assignments to location 3, and inserting the assignment of location 7 to location 1 in line 25. This assignment acts as the phi-node between the two predecessor blocks of block 1; namely block 0 and block 4.

```
1  bb0: {
2      _1 = &mut _2;                    // scope 0 at src/main.rs:8:20: 8:30
3      goto -> bb1;                     // scope 2 at src/main.rs:11:5: 16:6
4  }
5
6  bb1: {
7      _5 = &mut (*_1);                 // scope 2 at src/main.rs:12:11: 12:28
8      _4 = Thing::maybe_next(move _5) -> bb2; // scope 2 at src/main.rs:12:11: 12:28
9  }
10
11 bb2: {
12     _6 = discriminant(_4);           // scope 2 at src/main.rs:12:11: 12:28
13     switchInt(move _6) -> [0_isize: bb1, 1_isize: bb4, otherwise: bb3]; // scope 2 at src/main.rs:12:5:
       ↪  12:28
14 }
15
16 bb3: {
17     unreachable;                     // scope 2 at src/main.rs:12:11: 12:28
18 }
19
20 bb4: {
21     _7 = move ((_4 as Some).0: &mut Thing); // scope 2 at src/main.rs:13:14: 13:15
22     _1 = move _7;
23     goto -> bb1;                     // scope 2 at src/main.rs:13:29: 13:30
24 }
```

Code block 21: Modified MIR which `mir-owner-guillotine` can analyse

Note that the removal of the function type declarations makes us forget that location 1 is of type `&mut Thing`.

Passing the modified MIR in code block 21 into `mir-owner-guillotine` returns the following output:

```
bb 0 liveness:
        stmt:0 live in: {2}          live out: {1}
        stmt:1 live in: {1}          live out: {1}
==========
bb 1 liveness:
        stmt:0 live in: {1}          live out: {1, 5}
        stmt:1 live in: {1, 5}          live out: {1, 4}
==========
bb 2 liveness:
        stmt:0 live in: {1, 4}          live out: {1, 4, 6}
        stmt:1 live in: {1, 4, 6}          live out: {1, 4}
==========
bb 3 liveness:
        stmt:0 live in: set()          live out: set()
==========
bb 4 liveness:
        stmt:0 live in: {4}          live out: {7}
        stmt:1 live in: {7}          live out: {1}
        stmt:2 live in: {1}          live out: {1}
==========
Borrow-checking:
Bcking borrowee 1 with borrows:
        Borrow(er=5, ee=1, mut=True, bb1, s0)
Bcking borrowee 2 with borrows:
        Borrow(er=1, ee=2, mut=True, bb0, s0)
Bcking borrowee 4 with borrows:
        Borrow(er=7, ee=4, mut=True, bb4, s0)
borrow-check thinks program is valid? True

Process finished with exit code 0
```

Code block 22: Result of `mir-owner-guillotine` by running on loop-cond-mut MIR

E Trivially borrowing-wise faulty MIR program

```
bb0: {
    _1 = HashMap::new(const 42_u32); // make some var to ref to
    _2 = &(*_1); // immut ref to map for reading
    _3 = HashMap::<u32, String>::get::<u32>(move _2); // get val, semantics say result (loc 3) is now
    ↪  also a borrow of same type, to arg borrow (loc 2)
    _4 = &mut (*_1); // mut ref to map for writing
    _5 = ((_2 as Some).0: &std::string::String); // use of get-val, which generates liveness of borrow(3
    ↪  -> 1), but mut borrow above?!
    _0 = &mut (*_4); // finally use of mut ref to map, which is made live here
    return;
}
```

Code block 23: Hypothetical, hand-written MIR, which interleaves mutable and immutable borrows of the same location.

```
bb 0 liveness:
        stmt:0 live in: set()          live out: {1}
        stmt:1 live in: {1}          live out: {1, 2}
        stmt:2 live in: {1, 2}          live out: {1, 2}
        stmt:3 live in: {1, 2}          live out: {2, 4}
        stmt:4 live in: {2, 4}          live out: {4}
        stmt:5 live in: {4}          live out: {0}
        stmt:6 live in: {0}          live out: {0}
==========
Borrow-checking:
Bcking borrowee 1 with borrows:
        Borrow(er=2, ee=1, mut=False, bb0, s1)
        Borrow(er=3, ee=1, mut=False, bb0, s2)
        Borrow(er=4, ee=1, mut=True, bb0, s3)
checking         Borrow(er=2, ee=1, mut=False, bb0, s1)
and                 Borrow(er=3, ee=1, mut=False, bb0, s2)
both immutable, don't care about overlap

checking         Borrow(er=2, ee=1, mut=False, bb0, s1)
and                 Borrow(er=4, ee=1, mut=True, bb0, s3)
BCK ERROR: found overlap at b_i: 0, s_i: 4

borrow-check thinks program is valid? False
```

Code block 24: `mir-owner-guillotine` output on MIR in code block 23