

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Measurement



Master's Thesis

Memory Safety Analysis in Rust GCC

Jakub Dupák

Supervisor: Ing. Pavel Píša Ph.D.

Arthur Cohen

Study programme: Open Informatics
Branch of study: Computer Engineering

January 2024

I. Personal and study details

Student's name: **Dupák Jakub**

Personal ID number: **483785**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Open Informatics**

Specialisation: **Computer Engineering**

II. Master's thesis details

Master's thesis title in English:

Memory safety analysis in Rust GCC

Master's thesis title in Czech:

Analýza bezpečnosti paměti pro kompilátor Rust GCC

Guidelines:

Rust is a modern programming language focused on producing safe and performant code that is being largely adopted across the programming industry. The Rust compiler `rustc` is implemented on top of the LLVM compiler framework. GCCRS implements a new Rust front end on top of GCC to leverage GCC capabilities for Rust projects and provides a second independent Rust implementation.

The student will implement memory safety analysis (borrow checking) in the Rust GCC compiler using the Polonius project.

- 1) Study Polonius API and analysis principles.
- 2) Study Rust GCC control-flow information representation.
- 3) Design and implement foreign-function interface from Rust GCC (C++) to Polonius (Rust).
- 4) Design and implement input of control-flow information to Polonius.
- 5) Design and implement input of relevant memory operation facts to Polonius.
- 6) Design and implement output of Polonius analysis and basic error reporting.

Bibliography / sources:

- [1] MATSAKIS, Nicholas D. and KLOCK, Felix S., 2014, The rust language. ACM SIGAda Ada Letters. 2014. Vol. 34, no. 3, p. 103–104. DOI: 10.1145/2692956.2663188.
- [2] An alias-based formulation of the borrow checker, 2018. Baby Steps [online], Accessed June 2023. Available from: <http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker>
- [3] RAKIC, Rémy and MATSAKIS, Niko. Polonius. Available from: <https://rust-lang.github.io/polonius/>

Name and workplace of master's thesis supervisor:

Ing. Pavel Piša, Ph.D. Department of Control Engineering FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **01.09.2023**

Deadline for master's thesis submission: **09.01.2024**

Assignment valid until:

by the end of winter semester 2024/2025

Ing. Pavel Piša, Ph.D.
Supervisor's signature

Head of department's signature

prof. Mgr. Petr Páta, Ph.D.
Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Student's signature

Acknowledgements

”TODO”

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague on 8th January 2024

.....

Abstract

”XX”

Abstrakt

”XX”

Contents

1	Introduction	2
2	The Problem of Borrow-Checking	3
2.1	The Evolution of Borrow-Checking in rustc	5
3	Polonius Engine	8
3.1	Polonius Facts	9
4	Comparison of Internal Representations	12
4.1	GCC and LLVM	12
4.2	Rustc's Representation	13
4.3	Rust GCC representation	17
5	Rust GCC Borrow-checker Design	19
5.1	Analysis of the Fact Collection Problem	19
5.2	The Borrow-Checking Process	21
5.3	Representation of Lifetimes in TyTy	23
5.4	Borrow-checker IR Design	24
5.5	BIR Building	26
5.6	BIR Fact Collection and Checking	27
5.6.1	Subtyping and Variance	28
5.6.2	Variance of Generic Types	29
5.7	Error Reporting	31
6	Current State	33
6.1	Kind of Detected Errors	33
6.2	Parsing	33
6.3	AST to HIR Lowering	33
6.4	Type Checking (TyTy Representation)	33
6.5	Borrow-checker Scheduling	33
6.6	BIR Building	33
6.6.1	BIR Dump	34
6.7	BIR Fact Collection	34
6.8	Polonius FFI	34
6.9	Error Reporting	34
A	Rustc Intermediate Representations Examples	35
A.1	Rust source code	35
A.2	Abstract Syntax Tree (AST)	36
A.3	High-Level Intermediate Representation (HIR)	39
A.4	Mid-Level Intermediate Representation (MIR)	42

B	Comparison of BIR and MIR	43
B.1	Compilation Commands	43
B.2	Rust Source Code	43
B.3	BIR (Rustc GCC)	44
B.4	MIR (rustc)	45
C	Examples of Errors Detected by the Borrow-checker	48
C.1	Move Errors	48
C.2	Subset Errors	49
C.3	Loan Error	49
D	References	51

List of Figures

3.1	Steps performed by Polonius to find error. The starting nodes correspond to <i>facts</i> supplied to Polonius by the compiler. Inner nodes represent intermediate results produced by the analysis. Errors (at the bottom) are ultimately returned to the compiler. (The graphic was adapted from [5].)	10
4.1	LLVM IR CFG Example (generated by Compiler Explorer)	13
4.2	Comparison of compiler pipelines with a focus on internal representations	14
5.1	Placement of the borrow-checker IR in the compilation pipeline	22

Chapter 1

Introduction

The first chapter introduces the problem of borrow-checking and gives a brief overview of the borrow-checker development in the rustc compiler, up to the Polonius project, which is utilized by this work. The second chapter describes the Polonius analysis engine and its API. The third chapter compares the internal representations of rustc and gccrs to highlight the challenges of adapting the rustc borrow-checker design to gccrs. The next chapter explains the design of the borrow-checker implemented in gccrs as part of this work. It maps the experiments leading to the current design and describes the new intermediate representation and its usage in the analysis. Later sections of the chapter describe other modifications of the rest of the compiler necessary to support borrow-checking. The final chapter elaborates on the results, the current state of the implementations, and known missing features and limitations. Since this work had an experimental nature, it focused on exploring most aspects of the problem rather than on the completeness of the solution. Therefore, the final chapter should lead to future work, extending this experimental work into a production-ready solution.

Chapter 2

The Problem of Borrow-Checking

This section introduces borrow-checking and briefly describes its development in Rust. First, simple lexical borrow-checking is described. Then, the more complex control-flow sensitive borrow-checking is introduced. Finally, the Polonius analysis engine is described. Since this work utilizes the Polonius engine, it is described in more detail in the following chapter.

Typical programming language implementations fall into two categories based on how they manage memory with dynamic storage duration¹. Languages like C use manual memory management, where the programmer is responsible for allocating and freeing memory explicitly. Higher-level languages like Java or Python use automatic memory management, where a garbage collector² manages the memory in runtime. Since the C approach is considerably error-prone³, later languages like C++ and Zig³ provide tools to make the deallocation of memory more implicit. For simple cases, they tie the deallocation to the destruction of other objects (RAII⁴, smart-pointers⁵, defer statements⁶). What differentiates it from the stack allocation is that the relationship (called ownership) can be transferred dynamically between objects. For more complex cases, when there is no single object to which we can tie the deallocation (that means there are multiple objects, and the deallocation has to be linked to the destruction of the last one), they opt-in for a runtime solution (reference counting⁷). Those approaches improve the situation considerably. However, there are two problems remaining. First, those ownership bounds can be created incorrectly by the programmer, mainly in cases where the ownership of the memory is transferred between objects. The situation is even worse when two systems with different memory management models are interfaced⁸. Second, when the ownership is not transferred, but a copy of the pointer is used temporarily (this is called “borrowing” in Rust), assuming that the owning object will exist for the whole time this copy is used. Such an assumption is often wrong, and this kind of mistake is often called a “dangling pointer.”^[2]

The Rust language builds on the RAII approach; however, it adds a built-in static

¹Dynamic storage duration means that it is unknown at compile time when storage can be safely reclaimed. In contrast, memory with static duration is reclaimed at the end of the program, and memory with automatic storage duration is bound to a function call.

²[https://en.wikipedia.org/wiki/Garbage_collection_\(computer_science\)](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science))

³<https://ziglang.org/>

⁴<https://en.cppreference.com/w/cpp/language/raii>

⁵https://en.cppreference.com/w/cpp/memory/Smart_pointers

⁶<https://ziglang.org/documentation/master/#defer>

⁷https://en.wikipedia.org/wiki/Reference_counting

⁸An interface between a C++ application with STL-based⁹ memory management and the Qt GUI framework¹⁰, where all Qt API methods take raw pointers (as opposed to smart pointers.). Some of those methods assume that the ownership is transferred, and some of them do not. Those methods can only be differentiated using their documentation.

analysis called the borrow-checker to make sure that the mistakes mentioned above cannot happen. To make such analysis feasible, Rust only allows a conservative subset of memory-safe operations. Furthermore, Rust adds further limitations to ensure that memory use is safe even during multithreaded execution. Because these restrictions are very strict, and they would severely limit the language, Rust provides a feature to lift some of those restrictions in clearly denoted “unsafe” areas. The responsibility for maintaining the safety invariants in “unsafe” code falls on the programmer.

The key idea behind Rust memory safety is to strictly (using the type system) differentiate the two problematic cases: ownership transfers and borrows. Ownership transfer ties all owned unique resources to another object, detaching them from the current object. This operation is called “move” in Rust (and C++). Unlike C++, Rust does not allow objects to store a reference to themselves, simplifying the ownership transfer semantics to just a bitwise copy. Rust static analysis also ensures that the old object is not used after it has been “moved from.”

Borrow is a temporary usage of an object without ownership transfer. A typical example is a method call. For borrows, Rust uses static analysis to ensure that the borrowed object cannot be deallocated while in use (in Rust terms, the borrowed object has to *outlive* the borrow). However, since a whole program analysis would be very expensive, Rust performs the analysis only inside of a single function. It requires the programmer to formally describe the invariants of lifetimes (subsets of the program where each reference has to be valid) on a function boundary. The invariants are checked inside the function and assumed outside of the function, resulting in a safe program. The invariants are described using so-called lifetime annotations. The programmer can think about a lifetime annotation as an inference variable. The domain of the variable represents a subset of the program (set of lines, expressions control, or control flow graph nodes). The task of the borrow-checker is to resolve each inference variable to an actual subset of the program where the borrow is valid. This subset might not be unique. The existence of such a subset is sufficient to prove that the program is safe.

The annotations are related to each other by “outlives” relations, requiring one reference lifetime to be a subset of another lifetime. These constraints are used to describe the relationship of the inputs and outputs of a function, providing a simplified, conservative description of all the relevant code outside of the function.

Example: We have a vector-like structure (a dynamic array), and we want to store references to integers as elements. We need to make sure that as long as the vector exists, all references stored in it are valid. However, we do not want the vector to own the integers. First, we introduce a lifetime parameter 'a, which represents all the regions where the vector itself is alive. This parameter will be substituted at a particular use site with a concrete lifetime.

```
struct Vec<'a> { ... }
```

Then, for the add method, we introduce a lifetime parameter 'b, which restricts the inserted reference. This parameter is substituted with a concrete lifetime of each reference when the method is invoked. Finally, we will require that the method can only be used with lifetimes, for which we can guarantee that 'b is a subset of 'a (in terms of parts of the program). We do that by adding the 'a: 'b constraint. 'a: 'b usually reads as “'a outlives 'b,” and it means that “'a lasts at least as long as 'b, so a reference &'a i32 is valid whenever &'b i32 is valid.”^[3]^a

```
impl<'a> Vec<'a> {
    fn add<'b> where 'a: 'b (&mut self, x: &'b i32) { ... }
}
```

^a<https://doc.rust-lang.org/reference/trait-bounds.html#lifetime-bounds>

2.1 The Evolution of Borrow-Checking in rustc

This section describes how the analysis evolved, gradually rejecting less memory-safe programs. rustc started with lexical (scope-based analysis), followed by the first non-lexical (CFG-based) analysis, which is being extended by the Polonius project. This section strongly builds upon RFC 2094^[4], which introduced non-lexical borrow-checking to Rust. Examples from that RFC are presented in this section.

The simplest variant of borrow-checker is based on stack variable scopes. A reference is valid from the point in the program (here in terms of statements and expression) where it is created until the end of the current scope. This approach can be extended to handle some common programming patterns as special cases. For example, when a reference is created in function parameters, it is valid until the end of the function call.

```

{
    let mut data = vec!['a', 'b', 'c']; // --+ 'scope
    capitalize(&mut data[..]);           //   |
    // ~~~~~ 'lifetime //   |
    data.push('d');                      //   |
    data.push('e');                      //   |
    data.push('f');                      //   |
} // <-----+

```

However, a very common modification might cause the program to be rejected. Since the reference is not created in the list of function arguments but rather as a local variable, the special case does not apply, and the reference has to be valid until the end of the scope for the variable `slice`.

```

{
    let mut data = vec!['a', 'b', 'c'];
    let slice = &mut data[..]; // <-+ 'lifetime
    capitalize(slice);         //   |
    data.push('d'); // ERROR! //   |
    data.push('e'); // ERROR! //   |
    data.push('f'); // ERROR! //   |
} // <-----+

```

Now, there is no simple way to say when the lifetime of the reference should end to prove that his program is safe from its syntactic structure. The code above can be fixed by explicitly specifying where the lifetime should end. However, this clutters the code, and it cannot be used for more advanced cases.

```

{
    let mut data = vec!['a', 'b', 'c'];
    {
        let slice = &mut data[..]; // <-+ 'lifetime
        capitalize(slice);         //   |
    } // <-----+
    data.push('d'); // OK
    data.push('e'); // OK
    data.push('f'); // OK
}

```

One of those more advanced cases happens when lifetimes are not symmetric in conditional branches. A typical case is when a condition checks the presence of a value. In the positive branch, we are holding a reference to the value, but in the negative branch, we are not. Therefore, it is safe to create a new reference in the negative branch. By “safe,” we mean that there will only be one reference pointing to the `map` object at any time. A convenient way to describe “at any time” is using the control flow graph (CFG) representation of the program.

```

let mut map = ...;
let key = ...;
match map.get_mut(&key) { // -----+ 'lifetime
    Some(value) => process(value),      // |
    None => {                             // |
        map.insert(key, V::default()); // |
        // ~~~~~ ERROR.                 // |
    }                                    // |
} // <-----+

```

For more examples, see the RFC 2094[4]. However, the provided examples should be sufficient to demonstrate that analyzing the program on a control flow graph (CFG) instead of the syntactic structure (AST) enables the borrow-checker to validate and ensure the safety of complex programs that were previously rejected.

The above analysis thinks about lifetimes as regions (set of points in CFG) where the reference is valid. The goal of the analysis is to find the smallest regions such that the reference is not required to be valid outside of those regions. The smaller the regions, the more references can coexist at the same time, allowing more programs to be accepted.

The next generation of borrow-checking in Rust is based on the Polonius analysis engine. Polonius is an extension of NLL (non-lexical lifetimes), which is capable of proving move programs to be safe by using a different interpretation of lifetimes.

NLL cannot handle the following code, but Polonius can handle it. The problem here is that everything that is tied to external lifetimes ('a) has to be valid for the whole function. Since `v` is returned, it has to outlive the lifetime 'a. However, the lifetime of `v` is bound to the lifetime of the reference to the hashmap it is stored in. It forces the `map` to be borrowed (transitively) for at least the whole function. That includes the `map.insert` call, which needs to borrow the hashmap itself, resulting in an error. However, we can clearly see that no reference to `map` is available in the `None` branch. This is where Polonius can help.

Instead of starting with references and figuring out where they need to be valid, Polonius goes in the other direction and tracks what references need to be valid at each point in the program. As we have determined in the example above, there is no preexisting reference to the `map` in the `None` branch.

It is important to note that only internal computations inside the compiler are changed by this. This change does not affect the language semantics. It only lifts some limitations of the compiler.

Another significant contribution of the Polonius project is the fact that it replaces many handwritten checks with formal logical rules. Also, because it knows which references are conflicting, it can be used to provide better error messages.

Chapter 3

Polonius Engine

The Polonius engine was created by Niko Matsakis¹ and extended by Rémy Rakic² and Albin Stjerna[5] as a next-generation control-flow sensitive borrow-checking analysis for rustc. It was designed as an independent library that can be used both by the rustc compiler and different research projects, which makes it suitable for usage in gccrs. Polonius interfaces with the compiler by passing around a struct of vectors³ of facts, where each fact is represented by a tuple of integers⁴ (or types convertible to integers). It is completely unaware of the compiler internals.

In the previous chapter, we have mentioned that Polonius differs from NLL in its interpretation of lifetimes. Polonius uses the term “Origin” to better describe the concept. An origin is a set of loans that can be referenced using a variable at each CFG point. In other words, it tracks “where the references that are used could have originated.”

```
let r: &'0 i32 = if (cond) {
    &x /* Loan L0 */
} else {
    &y /* Loan L1 */
};
```

Example: *The origin of the reference r (denoted as $'0$) is the set of loans $L0$ and $L1$. Note that this fact is initially unknown, and it is the task of the analysis to compute it.*

The engine first preprocesses the input facts. It computes transitive closures of relations and analyzes all the initialization and deinitializations that happen over the CFG. Then, it checks for move errors, i.e., when ownership of some object is transferred more than once. In the next step, the liveness of variables and the “out-lives” graph (transitive constraints of lifetimes at each CFG point) are computed[6]. All origins that appear in the type of live variable are considered live.

Then Polonius needs to figure out *active loans*. A loan is active at a CFG point if two conditions hold. Any origin that contains the loan is live (i.e., there is a variable that might reference it), and the variable/place referencing the loan was not reassigned. (When a reference variable is reassigned, it points to something else.)

¹<https://github.com/nikomatsakis>

²<https://github.com/lqd/>

³A contiguous growable array type from the Rust standard library. (<https://doc.rust-lang.org/std/vec/struct.Vec.html>)

⁴`usize`

The compiler has to specify all the points in the control flow graph where a loan being alive would violate the memory safety rules. Polonius then checks whether such a situation can happen. If it can, it reports the facts involved in the violation. For example, if a mutable loan of a variable is alive, then any read/write/borrow operation on the variable invalidates the loan.

3.1 Polonius Facts

This section provides a list of facts taken by Polonius to give the reader a better idea of the work that needs to be done by the compiler. The facts are grouped into categories and briefly described. The full list of facts can be found in the Polonius source code⁵ and the Polonius Book[7].

- Control flow graphs edges (`cfg_edge: (Point, Point)`).
- Facts regarding variable usage and its effects.
 - `var_used_at: (Variable, Point)` - Any usage of variable except for a drop (destructor).
 - `var_defined_at: (Variable, Point)` - Start of scope or reassignment. This reassignment treatment makes the variable act similarly to an SSA variable⁶.
 - `var_dropped_at: (Variable, Point)` - Drop (destructor call) of the variable.
 - `use_of_var_derefs_origin: (Variable, Origin)` - The type of the variable contains the origin.
 - `drop_of_var_derefs_origin: (Variable, Origin)` - When the drop implementation used the origin.
- Facts regarding paths and their usages. Paths represent indirect or partial access to a variable (e.g., field access or cast).
 - `path_is_var: (Path, Variable)` - Lists “trivial” paths that are just a variable.
 - `child_path: (Path, Path)` - Describes hierarchical (non-transitive) relationships between paths. For example, a field path is a child path of the variable path it is accessed from.
 - `path_assigned_at_base: (Path, Point)` - Path is assigned at the CFG point. “base” means that this fact is emitted only for the exact path used, not all its parent paths.
 - `path_moved_at_base: (Path, Point)` - Ownership of origins is transferred at the CFG point.
 - `path_accessed_at_base: (Path, Point)` - Any memory access to the path (read or write).
- Facts about relationships (subset relation) of origins.
 - `known_placeholder_subset: (Origin, Origin)` - Constraints on universal origins (those representing loans that happened outside the function).
 - `universal_region: (Origin)` - List of universal origins. (See the previous point.)

⁵<https://github.com/rust-lang/polonius/blob/master/polonius-engine/src/facts.rs>

⁶https://en.wikipedia.org/wiki/Static_single-assignment_form

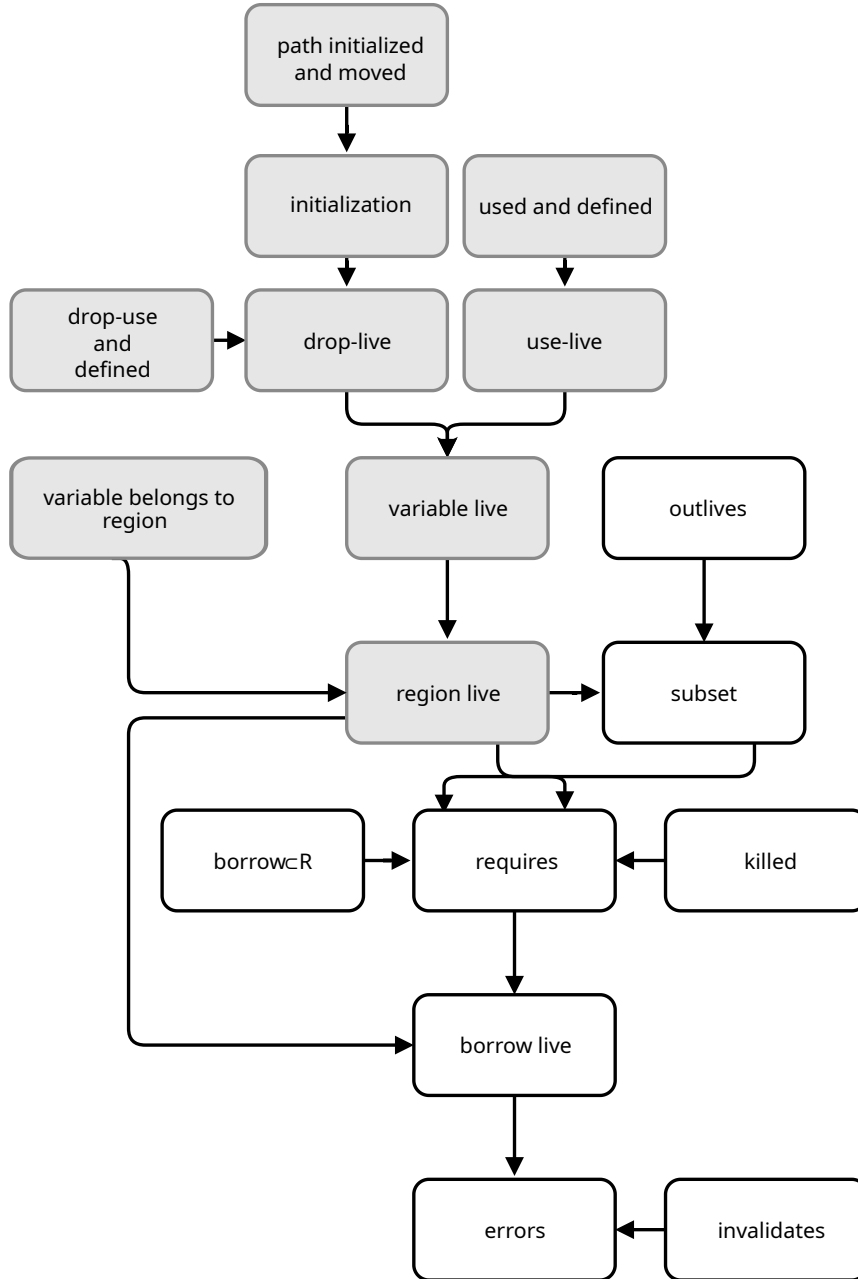


Figure 3.1: Steps performed by Polonius to find error. The starting nodes correspond to *facts* supplied to Polonius by the compiler. Inner nodes represent intermediate results produced by the analysis. Errors (at the bottom) are ultimately returned to the compiler. (The graphic was adapted from [5].)

3. Polonius Engine

- `subset_base: (Origin, Origin)` - Any relationship between origins required by the subtyping rules.
 - `placeholder: (Origin, Loan)` - Associates an origin with a loan.
- Facts about loans.
 - `loan_issued_at: (Loan, Point)` - Result of borrow expression.
 - `loan_killed_at: (Loan, Point)` - Loan is no longer live after this point.
 - `loan_invalidated_at: (Loan, Point)` - If the loan is live at this point, it is an error.

Chapter 4

Comparison of Internal Representations

The execution of a borrow-checker with an external analysis engine consists of two steps. First, we need to collect the relevant information about the program. We will call that information *facts*¹. Second, we need to send those facts to the external engine and process them. Before we can discuss the *collection* of facts itself, we need to understand how programs are represented inside the compiler. We will use the term *internal representation* (IR) to refer to the representation of the program inside the compiler. We will compare the IRs used by rustc and gccrs to highlight the differences between the two compilers. This will help us understand the challenges of adapting the borrow-checker design from rustc to gccrs. First, we will describe the IRs used by rustc, and then we will compare them with those used in gccrs.

4.1 GCC and LLVM

To understand the differences between each of the compilers, we must first explore the differences between the compiler platforms they are built on (GCC and LLVM). We will only focus on the middle-end of each platform since the back-end does not influence the front-end directly.

The core of LLVM is a three-address code (3-AD)² representation, called the LLVM intermediate representation (LLVM IR) [8], `llvm-ir`. This IR is the interface between front-ends and the compiler platform (the middle-end and the back-end). Each front-end is responsible for transforming its custom AST IR³ into the LLVM IR. The LLVM IR is stable and strictly separated from the front-end, hence it cannot be easily extended to include language-specific constructs.

GCC, on the other hand, interfaces with the front-ends using a tree-based representation called GENERIC[9, p. 175]. GENERIC was created as a generalized form of AST shared by most front-ends. GCC provides a set of common tree nodes to describe all the standard language constructs in the GENERIC IR. Front-ends may define language-specific constructs and provide hooks for their handling.[9, p. 212] This representation is then transformed into the GIMPLE representation, which is mostly⁴ a 3-AD representation. It does so by breaking down expressions into a

¹This follows the terminology used by Polonius[7].

²Three-address code represents the program as sequences of statements (we call such sequence a *basic block*), connected by control flow instructions, forming a control flow graph (CFG).

³Abstract syntax tree (AST) is a data structure used to represent the structure of the program. It is the direct product of program parsing. For example, an expression `1 + (2 - 7)` would be represented as a node `subtraction`, with the left child being the number one and the right child being the AST for the subexpression `(2 - 7)`.

⁴GIMPLE that is not fully lowered is known as “High GIMPLE” and consists of the IL before the `pass_lower_cf`. High GIMPLE contains some container statements like lexical scopes and nested expressions, while “Low GIMPLE” exposes all of the implicit jumps for control and exception

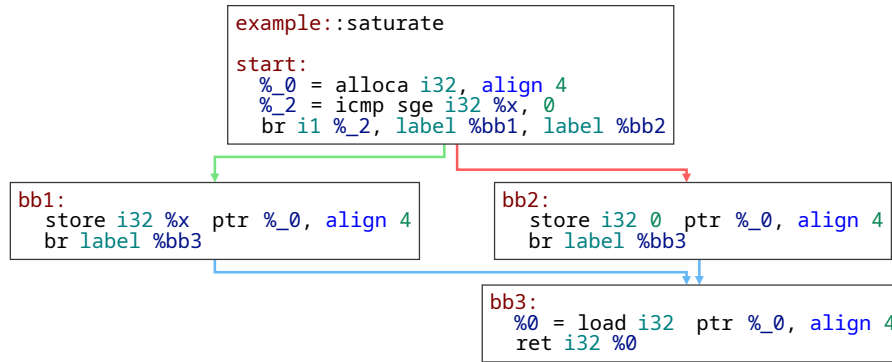


Figure 4.1: LLVM IR CFG Example (generated by Compiler Explorer)

sequence of statements and introducing temporary variables. This transformation is done inside the compiler platform, not in the front-end. This approach makes the front-ends smaller and shifts more work into the shared part. The GIMPLE representation does not contain information specific to each front-end (programming language). However, it is possible to store language-specific information in GIMPLE by adding entirely new statements.[9, p. 262] This is possible because GIMPLE is not a stable interface.

The key takeaway from this section is that rustc has to transform the tree-based representation into a 3-AD representation by itself. That means it can access the program’s control flow graph (CFG). This is not the case for gccrs. In GCC, the CFG is only available in the *Low GIMPLE* representation, deep inside the middle-end, where the representation is language independent.

4.2 Rustc’s Representation

In the previous section, we have seen that rustc is responsible for transforming the code from the raw text to the LLVM IR. Given the high complexity of the Rust language, rustc uses multiple intermediate representations (IRs) to simplify the process (see the diagram below). The text is first tokenized and parsed into an abstract syntax tree (AST) and then transformed into the high-level intermediate representation (HIR). For transformation into a middle-level intermediate representation (MIR), the HIR is first transformed into a typed HIR (THIR). The MIR is then transformed into the LLVM IR.

AST is a tree-based representation of the program, closely following each source code token. At this stage, rustc performs macro-expansion and a partial name resolution (macros and imports) [10]⁵⁶. As the AST is lowered to HIR, some complex language constructs are desugared to simpler constructs. For example, various kinds of loops are transformed into a single infinite loop construct (Rust loop keyword), and many structures that can perform pattern matching (if let,

expressions directly in the IL and EH region trees.”[9, p. 225]

⁵<https://rustc-dev-guide.rust-lang.org/macro-expansion.html>

⁶<https://rustc-dev-guide.rust-lang.org/name-resolution.html>

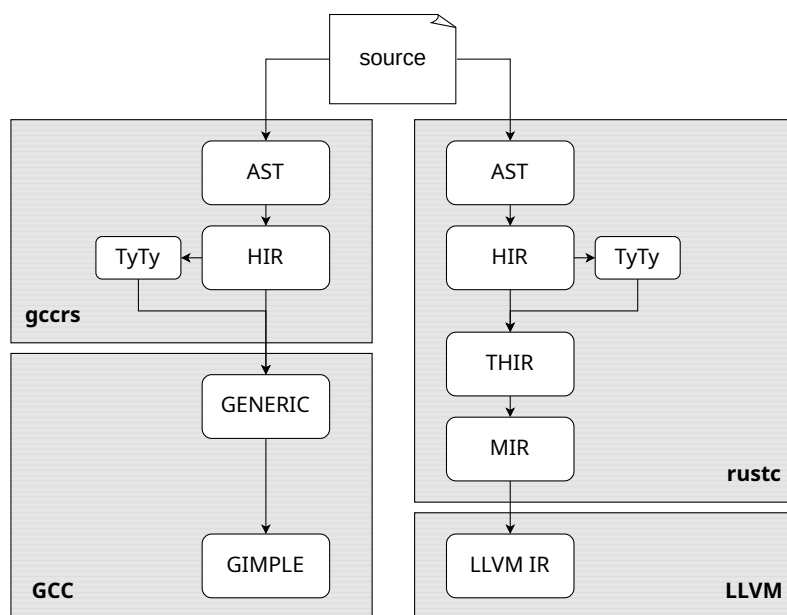


Figure 4.2: Comparison of compiler pipelines with a focus on internal representations

`while let, ? operator)` are transformed into the ‘match’ construct^[3]⁷.

```

struct Foo(i31);

fn foo(x: i31) -> Foo {
    Foo(x)
}

```

Example: This very simple code will be used as an example throughout this section.

⁷<https://doc.rust-lang.org/reference/expressions/if-expr.html#if-let-expressions>⁸

```

Fn {
  generics: Generics { ... },
  sig: FnSig {
    header: FnHeader { ... },
    decl: FnDecl {
      inputs: [
        Param {
          ty: Ty {
            Path { segments: [ PathSegment { ident: i32#0 } ] }
          }
          pat: Pat { Ident(x#0) }
        },
      ],
      output: Ty {
        Path { segments: [ PathSegment { ident: Foo#0 } ] }
      }
    },
  },
  body: Block {
    stmts: [ Stmt { Expr {
      Call(
        Expr {
          Path { segments: [ PathSegment { ident: Foo#0 } ] }
        }
        params: [
          Expr {
            Path { segments: [ PathSegment { ident: x#0 } ] }
          }
        ]
      )
    } } ]
  }
}

```

Example: This is a textual representation of a small and simplified part of the abstract syntax tree (AST) of the example program. The full version can be found in the Appendix A.

HIR is the primary representation used for most rustc operations[10], HIR. It combines a simplified version of the AST with additional tables and maps for quick access to extra information. Those tables contain, for example, information about the types of expressions and statements. These tables are used for analysis passes, e.g., the full (late) name resolution and type checking. The type-checking process includes checking the type correctness of the program, type inference, and resolution of type-dependent implicit language constructs.[10]⁹

⁹<https://rustc-dev-guide.rust-lang.org/type-checking.html>

```
#[prelude_import]
use ::std::prelude::rust_2015::*;
#[macro_use]
extern crate std;
struct Foo(i32);

fn foo(x: i32) -> Foo { Foo(x) }
```

Example: One of HIR dump formats: HIR structure still corresponds to a valid Rust program, equivalent to the original one. `rustc` provides a textual representation of HIR, which displays such a program.

The HIR representation can contain many placeholders and “optional” fields that are resolved during the HIR analysis. To simplify further processing, parts of HIR that correspond to executable code (e.g., not type definitions) are transformed into THIR (Typed High-Level Intermediate Representation), where all the missing information must be resolved. The reader can think about HIR and THIR in terms of the builder pattern¹⁰. HIR provides a flexible interface for modification, while THIR is the final immutable representation of the program. This involves not only the data stored in HIR helper tables but also parts of the program that are implied from the type system. That means that operator overloading, automatic references, and automatic dereferences are all resolved into explicit code at this stage.

The final `rustc` IR, which is lowered directly to the LLVM IR, is the Mid-level Intermediate Representation (MIR). We will pay extra attention to MIR because it is the primary representation used by the borrow-checker. MIR is a three-address code representation, similar to LLVM IR but with Rust-specific constructs. It contains information about types, including lifetimes. It differentiates pointers and references, as well as mutable and immutable references. It is aware of panics and stack unwinding. It contains additional information for borrow-checker, like storage live/dead annotations, which denote when a place (an abstract representation of a memory location) is first used or last used, and fake operations, which help with the analysis. For example, a fake unwind operation inside infinite loops ensures an exit edge in the CFG. Fake operations can be critical for algorithms that process the CFG in reverse order.

MIR consists of sequences of statements (basic blocks) connected by control flow instructions. This structure forms a control flow graph. MIR statements operate on places (often called lvalue in other languages) and rvalues. A place can represent either a local variable or a value derived from the variable (e.g., a field, an index, cast).

Rustc also uses a special IR, called the TyTy, to represent types. Initially, types are represented in HIR on a syntactic level. Every mention of a type in the program compiles into a distinct HIR node. These HIR nodes are compiled into the TyTy representation during the analysis of HIR. Each type (all of its occurrences in the program) is represented by a single TyTy object instance. This is achieved by interning¹¹. Note that there can be multiple equivalent types of different structures.

¹⁰https://en.wikipedia.org/wiki/Builder_pattern

¹¹https://en.wikipedia.org/wiki/Interning_%28computer_science%29

Those are represented by different TyTy instances. Each non-primitive type forms a tree (e.g., reference to a pair of an integer and a character), where the inner nodes are shared between types due to interning. Generic types, which are of particular interest to borrow-checking, are represented as a pair: an inner type and a list of generic arguments. When generic type parameters are substituted for concrete types, the concrete type is placed into the argument list. The inner type is left unchanged. When the type substitution is complete, there is a procedure that transforms the generic type into a concrete type.

Inside the HIR, after the type-checking analysis, TyTy types of nodes can be looked up based on the node's ID in one of the helper tables (namely, the type-check context). Each THIR node directly contains a pointer to its type. In MIR, the type is stored inside each place.

```
fn foo(_1: i32) -> Foo {
    debug x => _1;
    let mut _0: Foo;

    bb0: {
        _0 = Foo(_1);
        return;
    }
}
```

Example: MIR dump For further details, see the chapter “Source Code Representation” in [10].

4.3 Rust GCC representation

This section discusses intermediate representations in gccrs. Since gccrs is a second implementation of the Rust compiler, it is heavily inspired by rustc. Therefore, this section assumes familiarity with rustc's intermediate representations, described in the previous section. We will focus on similarities and differences between rustc and gccrs, rather than describing the gccrs intermediate representation in full detail.

The gccrs representation is strongly inspired by rustc. It diverges mostly for two reasons: for simplicity, since gccrs is still in an early stage of development, and due to the specifics of the GCC platform. Gccrs uses its own variants of AST, HIR, and TyTy representations but does not use a THIR or MIR.

AST and HIR representations are similar to rustc, with fewer features supported. The main difference is the structure of the representation. Rustc takes advantage of algebraic data types, resulting in very fine-grained representation. On the other hand, gccrs is severely limited by the capabilities of C++11 and is forced to use an object-oriented approach.

There is no THIR and MIR or any equivalent in gccrs. MIR cannot be used in GCC unless the whole gccrs code generation is rewritten to output (low) GIMPLE

instead of `GENERIC`, which would be way more complex than the current approach. Given the limited development resources of `gccrs`, this is not a viable option.[11]

TyTy type representation is simplified in `gccrs` and provides no uniqueness guarantees. There is a notable difference in the representation of generic types. Instead of being built on top of the types (by composition) like in `rustc`, types that support generic parameters inherit from a common base class. That means the type definition is not shared between different generic types. The advantage of this approach is that during the substitution of generic parameters, the inner types are modified during each type substitution, simplifying intermediate handling, like type inference.

Chapter 5

Rust GCC Borrow-checker Design

The Rust GCC borrow-checker is designed to be as similar to the rustc borrow-checker as possible within the constraints of the Rust GCC. This allows us to leverage existing knowledge about borrow-checking in Rust. The analysis works in two phases. First, it collects relevant information (called facts) about the program, which are stored as tuples of numbers. Each number represents a CFG node, variable, path/place, or loan (a borrow expression). Then, the borrow-checker passes the facts to the analysis engine, which computes the results of the analysis. The compiler receives back the facts involved in memory safety violations and translates them into error messages. The main decision of the Rust GCC borrow-checker is to reuse the analysis engine from rustc. To connect the Polonius engine written in Rust to the gccrs compiler written in C++, we use the C ABI and a thin Rust wrapper.

This chapter describes the process of designing the gccrs borrow-checker, the decisions made during the process, and the final design. Special emphasis is placed on a new borrow-checker intermediate representation (BIR) and its usage in the analysis. The chapter also describes other modifications of the compiler necessary to support borrow-checking. The final section briefly describes the design of error reporting.

5.1 Analysis of the Fact Collection Problem

This section described options for fact collection in gccrs that were considered and experimented with during the initial design phase. Due to the differences between internal representations of rustc and gccrs, it was impossible to copy the rustc approach exactly. Considered options were to use HIR directly, to implement MIR in gccrs, or to design a new IR for borrow-checking with multiple options to place it inside the compilation pipeline.

The analysis has been control-flow sensitive since NLL's introduction in rustc (see section 2.1), requiring us to match the required facts, which are specific to Rust semantics, with control-flow graph nodes. We need to distinguish between pointers (in unsafe Rust) and references. Pointers are not subject to borrow-checking, but references are. Furthermore, we need to distinguish between mutable and immutable references since they have different rules, which is essential for borrow-checking¹. Each type must carry information about its lifetimes and their variances (described later in this chapter). We need to store the explicit lifetime parameters from explicit user type annotation.

¹The critical rule of borrow-checking is that for a single borrowed variable, there can only be a single mutable borrow or only immutable borrows valid at each point of the CFG.

The only IR in GCC that contains CFG information is GIMPLE; however, under normal circumstances, GIMPLE is language agnostic. It is possible to annotate GIMPLE statements with language-specific information using special statements, which would have to be generated from special information that would need to be added to GENERIC. The statements would need to be preserved by the middle-end passes until the pass building the CFG (that includes 11 passes), after which facts could be collected. After that, the facts would need to be discarded to avoid complicating the tens of following passes²[9, p. 141], and RTL generation. This approach was discussed with senior GCC developers and quickly rejected as it would require a large amount of work and leak front-end-specific information into the middle-end, making it more complex. No attempt was made to experiment with this approach.

It was clear that we needed to build a CFG. Luckily, working with a particular control flow graph created by the compiler is unnecessary. Any CFG that is consistent with Rust semantics is sufficient. In particular, adding any edges and merging nodes in the CFG is conservative with regard to the borrow-checking analysis. In many cases, it does not change the result at all.

Initially, we tried to collect information from the HIR directly and compute an approximate CFG on the fly. That worked nicely for simple language constructs that are local, but it gets very complicated for more complex constructs like patterns and loops with `break` and `continue` statements. Since no representation is generated, there is no easy way to verify the process, not even by manual checking. Furthermore, it was not clear how to handle panics and stack unwinding in this model.

An option to ease such problems was to radically desugared the HIR to only basic constructs. An advantage of this approach is that it would leverage the code already existing in the code generator, making the code generation easier. Also, the code generator already performs some of those transformations locally (not applying them back to HIR, but using them directly for GENERIC generation), so those could be reused. The problem that quickly arose was that the HIR visitor system was not designed for HIR-to-HIR transformations, where new nodes would be created. Many such transformations, like explicit handling of automatic referencing and dereferencing, would require information about the type of each node, which would, in return, require name resolution results. Therefore, that transformation would have to happen after all analysis passes on HIR are completed. However, all information stored alongside HIR would need to be updated for each newly created node. The code generator partly avoids this problem by querying the GENERIC API for the information it needs about the already compiled code. This fact would complicate leveraging the existing transformations on the HIR to HIR level. Rustc avoids this problem by doing such transformations on the HIR-THIR boundary and not modifying the HIR itself. Since this modification would be complicated and would only be a preparation for borrow-checking, it was decided not to proceed in this direction at that time. However, we found that some transformation can be done on the AST-HIR boundary. This approach can be done mostly independently (only code handling the removed nodes is also removed, but no additions or modifications are needed). It was agreed that such transformations are useful and should be

²See file `gcc/passes.def` in the GCC source code.

implemented regardless of the path taken by the borrow-checker. Those transformations include mainly loops and pattern-matching structures. Those transformations are even documented in the rust reference[3].

At the time of writing, desugaring of the for loop was implemented by Philip Herron. More desugaring work is in progress or planned. However, I have focused on borrow-checking itself. For the time being, I have ignored the complex constructs, assuming that they will be eventually desugared to constructs that the borrow-checker would already be able to handle.

To ensure all possible approaches were considered, we have discussed the possibility of implementing MIR in gccrs. This approach has some advantages and many problems. Should the MIR be implemented in a completely compatible way, it would be possible to use tools like MIRI³ with gccrs. The borrow-checking would be very similar to rustc's borrow-checking, and parts of rustc's code might even be reused. Gccrs would also be more ready for Rust-specific optimizations within the front-end. The final advantage is that the current test suite would cover the process of lowering HIR to MIR, as all transformations would affect the code generation. The main problem with this approach is that it would require a large portion of gccrs to be reimplemented, delaying the project by a considerable amount of time. Should such an approach be taken, any effort on borrow-checking would be delayed until the MIR is implemented. The maintainers[11] decided that such an approach is not feasible and that gccrs will not use MIR in any foreseeable future.

After Arthur Cohen suggested keeping things simpler, I decided to experiment with a different, minimalistic approach—to build a radically simplified MIR-like IR that keeps only the bare minimum of information needed for borrow-checking. Given the unexpected productivity of this approach, it was decided to go on with it. This IR, later called the borrow-checker IR (BIR), only focuses on the flow of data, and it ignores the actual data transformations. The main disadvantage of this approach is that it creates a dead branch of the compilation pipeline that is not used for code generation, and therefore, it is not covered by the existing test suite. To overcome this difficulty, the BIR and its textual representation (dump) are designed to be as similar to rustc's MIR as possible. This feature allows us to check the generated BIR against the MIR generated by rustc, at least for simple programs. Using BIR is the final approach used in this work. Details of the BIR design are described in the next section.

5.2 The Borrow-Checking Process

Before the borrow-checking itself can be performed, specific information about types needs to be collected when HIR is type-checked and TyTy types are created and processed. The TyTy needs to resolve and store information about lifetimes and their constraints. At this point, lifetimes are resolved from string names, and their bounding clauses are found. There are different kinds of lifetimes in the Rust language. Inside types, the lifetimes are bound to the lifetime parameters of generic types. In function pointers, lifetimes can be universally quantified (meaning that the

³<https://github.com/rust-lang/miri>

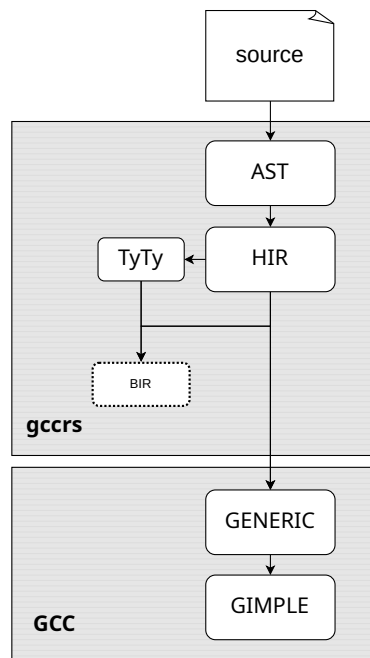


Figure 5.1: Placement of the borrow-checker IR in the compilation pipeline

function must be memory-safe for every possible lifetime). In function definitions, lifetimes can be elided when all references have the same lifetime. In function bodies, lifetimes can be bound to the lifetime parameters of the function, or they can be omitted, in which case they are inferred ⁴. The type-checked HIR is then transformed into the borrow-checker IR (BIR). The BIR is then processed to extract facts for Polonius. At this phase, some errors that are easy to detect can be emitted. The collected facts are then passed to Polonius, which computes the results of the analysis. The results are then passed back to the compiler, which translates them into error messages.

⁴At least Rust semantics thinks about it that way. In reality, the compiler only checks that there exists some lifetime that could be used in that position by collecting constraints that would apply to such a lifetime and passing them to the borrow-checker.

5.3 Representation of Lifetimes in TyTy

The term *lifetime* is used in this work to refer to the syntactic object in HIR and AST. In the source code it corresponds to either explicit universal^a lifetime annotation ('a), elided universal lifetime annotation[3]^b, and local/existential^c lifetimes, which are always inferred. In contrast, *region/origin* is used to refer to the semantic object. The object is in fact an inference variable, and its value is computed by the borrow-checker. The term *region* is used by NLL to refer to a set of CFG points. Polonius introduced the term *origin* to refer to a set of *loans*. In this text and the implementation, we use the two terms interchangeably.

^aThere are two kinds of lifetimes in Rust semantics: universal and existential. Universal lifetimes correspond to code that happens outside the function. It is called universal because the concerned borrow-checking rules use the universal quantifier. That means that the function has to be valid *for all* possible outside code that satisfies the specified (or implied) constraints. Existential lifetimes correspond to code that happens inside the function. The existential quantifier is used in the rules regarding existential lifetimes. That means that the code has to be valid *for some* set of *loans* (or CFG points).

^b<https://doc.rust-lang.org/reference/lifetime-elision.html>

^cThere are two kinds of lifetimes in Rust semantics: universal and existential. Universal lifetimes correspond to code that happens outside the function. It is called universal because the concerned borrow-checking rules use the universal quantifier. That means that the function has to be valid *for all* possible outside code that satisfies the specified (or implied) constraints. Existential lifetimes correspond to code that happens inside the function. The existential quantifier is used in the rules regarding existential lifetimes. That means that the code has to be valid *for some* set of *loans* (or CFG points).

In order to analyze more complex lifetimes than just simple references, it was necessary to add representation of lifetime parameters to the type system and unify it with the representation of lifetimes in the rest of the compiler. The first step is to resolve the lifetimes and bind them to their bounding clauses. Gccrs recognizes four kinds of regions. In a function body, explicit lifetimes annotations result in “named” lifetime and implicit lifetimes annotations result in “anonymous” lifetimes. Within generic data types lifetimes resolved to lifetime parameters are called “early-bound.” For function pointers and traits, lifetimes can be universally quantified using the `for` clause⁵. Those lifetimes are not resolved when the definition is analyzed but only when this type is used. Hence, the name is “late-bound” lifetimes. In addition, there is a representation for unresolved lifetimes. It is used, for example, when a generic type is defined, but the generic arguments have not been provided yet. Any occurrence of an unresolved lifetime after type checking it to be treated as a compiler bug.

Inside TyTy, lifetimes are represented in the following ways. Named lifetimes are enumerated. Anonymous lifetimes are assumed to be always distinct (but they are represented by an identical object at this stage). Early bound lifetimes are represented by the relative position of the lifetime parameter to which they are bound. In generic types, the lifetime arguments are stored together with the type arguments, which ensures their automatic propagation. One issue with this automatic propagation is that the bindings of early bound lifetimes are updated. That means

⁵`for<'a> fn(&'a i32) -> &'a i32`

that by a simple inspection of the body of the generic type, one would not be able to resolve the lifetimes. A trick resolves this problem. Each TyTy type is identified by an ID. When generic arguments are substituted, a clone of the type with a fresh ID is created. What we would like to achieve is to have the same state that is in rustc: the original body and an up-to-date list of generic arguments. This can be achieved by storing the ID of the original type in addition to the current ID. The original ID can be used to lookup the original type when needed.⁶ The analysis can then traverse the original type, and when a type placeholder is encountered, the appropriate argument is looked up in the current type.

5.4 Borrow-checker IR Design

The borrow-checker IR (BIR) is a three-address code representation designed to be close to a subset of rustc MIR. Like MIR, it represents the body of a single function (or a function-like item, e.g., a closure) since borrow-checking is performed on each function separately. It ignores particular operations and merges them into a few abstract operations focusing on data flow.

```
fn fib(_1: usize) -> i32 {
  bb0: {
    _4 = Operator(_1, const usize);
    switchInt(_4) -> [bb1, bb2];
  }

  // ... (rest of the code)

  bb6: {
    _8 = Operator(_1, const usize);
    _9 = Call(fib)(_8, ) -> [bb7];
  }

  bb7: {
    _10 = Operator(_7, _9);
    _2 = _10;
    goto -> bb8;
  }

  bb8: {
    _0 = _2;
    return;
  }
}
```

⁶This was once revealed to me in a dream.

Example: A shortened example of a BIR dump of a simple Rust program. The program computes the n -th Fibonacci number. The source code, full dump, and legend can be found in *appendix C*. This example comes from the “BIR Design Notes” document, which is part of the source tree and which provides an introduction to developers getting familiar with the basic aspects of the borrow-checker implementation.

The BIR of a single function is composed of basic metadata about the function (such as arguments, return type, or explicit lifetimes), a list of basic blocks, and a list of places.

A basic block is identified by its index in the function’s basic block list. It contains a list of BIR statements and a list of successor basic block indices in the CFG. BIR statements are of three categories: An assignment of an expression to a local (place), a control flow operation (switch, return), or a special statement (not executable), which carries additional information for the borrow-checker (explicit type annotations, information about variable scope, etc.). BIR statements correspond to the `MIR StatementKind` enum.

Expressions represent the executable parts of the rust code. Many different Rust constructs are represented by a single expression. Only data (and lifetime) flow needs to be tracked. Some expressions are differentiated only to allow for a better debugging experience. BIR expressions correspond to the `MIR RValue` enum.

Expressions and statements operate on places. A place is an abstract representation of a memory location. It is either a variable, a field, an index, or a dereference of another place. For simplicity, constants are also represented as places. Since exact values are not important for borrow-checking and constants are, from principle, immutable with static storage duration, a single place can represent all constants of a single type. Rustc MIR cannot afford this simplification, and it keeps constants separate. The `Operand` enum is a common interface for places and constants. However, since operations use constants and lvalues the same way, MIR introduces a special layer of lvalues.

Places are identified by the index in the place database. The database stores a list of places and their properties. The properties include an identifier, used to always resolve the same variable (field, index, etc.) to the same place, move and copy flags, type, a list of fresh regions (lifetimes), and a relationship to other places (e.g., a field of a struct). Temporaries are treated just like variables but are differentiated in the place database because of place lookup. The place database also keeps track of scopes and existing loans. The place database structure is based on rustc `MovePathData`⁷. It combines the handling of places done by both MIR and borrow-checker separately in rustc.

It is important to highlight that different fields are assigned to different places; however, all indices are assigned to the same place (both in gccrs and rustc). This fact has a strong impact on the strength and complexity of the analysis; since the number of fields is static and typically small, the size of arrays is unbound and

⁷https://rustc-dev-guide.rust-lang.org/borrow_check/moves_and_initialization/move_paths.html

depends on runtime information.

Structure of BIR Function

- basic block list
- Statement
 - Assignment
 - * `InitializerExpr`
 - * `Operator<ARITY>`
 - * `BorrowExpr`
 - * `AssignmentExpr` (copy)
 - * `CallExpr`
 - Switch
 - Goto
 - Return
 - `StorageLive` (start of variable scope)
 - `StorageDead` (end of variable scope)
 - `UserTypeAsscription` (explicit type annotation)
- place database
- arguments
- return type
- universal lifetimes
- universal lifetime constraints

5.5 BIR Building

The BIR is built by visiting the HIR tree of the function. There are specialized visitors for expressions and statements, patterns, and a top-level visitor that handles function headers (arguments, return, lifetimes, etc.). Whenever a new place is created in the compilation database, a list of fresh regions⁸ is created for it. At this point, we need to figure out the number of lifetimes mentioned in a type. For basic types, this is achieved by traversing the type and counting the number of lifetime parameters. For generic types, the inner structure is ignored, and only the lifetime and type parameters are considered. Note that the type parameters can be generic, creating a structure known as higher-kinded⁹ lifetimes. This counting is performed (as a side product) during the variance analysis (explained below) to simplify the type traversing code. All types are independently queried for each node from HIR (they are not derived inside the BIR).

Example: For a BIR code that reads a field from a variable, the type is not computed from the variable. Rather, it is queried from the HIR for both the variable and the field.

⁸In this text, we use the term lifetime for the syntactic object in the code and region for the semantic object in the analysis. It is called a region because it represents a set of points in the control flow graph (CFG). At this point, the set is not yet known. It is the main task of the borrow-checker analysis engine to compute the set of points for each region.

⁹<https://rustc-dev-guide.rust-lang.org/what-does-early-late-bound-mean.html#early-and-late-bound-parameter-definitions>

BIR building itself is fairly straightforward. However, some extra handling was added to produce a code that is more similar to `rustc`'s MIR. For example, instead of eagerly assigning computed expressions to temporaries, it is checked whether the caller did not provide a destination place. This transformation removes some of the `_10 = _11` statements from the BIR dump. The BIR dump also renumbers all places to produce a closer match with the BIR dump. This can cause some confusion during debugging because Polonius is receiving the original place numbers. When debugging using the Polonius debug output, the dump can be switched to show the original place numbers.

This handling was especially important when testing the initial BIR builder since it makes the dump more similar to the MIR dump and, therefore, easier for manual comparison.

5.6 BIR Fact Collection and Checking

The BIR fact collection extracts the Polonius facts from the BIR and performs additional checks. Polonius is responsible for checking lifetime (region) constraints, moves, and conflicts between borrows. For lifetimes, it checks that the constraints are satisfied and that all required constraints are present in the program. For moves, it checks that each place is moved at most once. For borrows, it checks that any two conflicting borrows (e.g., two mutable borrows of the same place) are not alive at the same time. Sets of conflicting borrows have to be supplied to Polonius manually. The borrow-checker itself is responsible for violations that are not control-flow sensitive, like modification of an immutably borrowed place or moving from behind a reference.

The fact collection is performed in two phases. First, static facts are collected from the place database. Those include universal region constraints (constraints corresponding to lifetime parameters of the function) collected during BIR construction and facts collected from the place database. Polonius needs to know which places correspond to variables and which form paths (see definition below). Furthermore, it needs to sanitize fresh regions of places that are related (e.g., a field and a parent variable) by adding appropriate constraints between them. Relations of the region depend on the variance of the region within the type. (See Variance Analysis below.)

```
Path = Variable
    | Path "." Field // field access
    | Path "[" "]"   // index
    | "*" Path
```

Formal definition of paths from the Polonius book[7].

In the second phase, the BIR is traversed along the CFG, and dynamic facts are collected. For each statement, two CFG nodes are added. Two nodes are needed to model the parts of semantics where the statement takes effect immediately or after the statement is executed. For each statement and (if present) its expression, Polonius facts are collected. Those include generic facts related to read and write

operations, as well as facts specific to borrows and function calls. For the function, we need to instantiate fresh regions for the function’s lifetime parameters, which need to be correctly bound together.

5.6.1 Subtyping and Variance

In the basic interpretation of Rust language semantics (one used by programmers to reason about their code, not the one used by the compiler), lifetimes are part of the type and are always present. If a lifetime is not mentioned in the program explicitly, it is inferred the same way as a part of type would be (e.g., `let a = (_, i32) = (true, 5);` completes the type to `(bool, i32)`) Note that it is actually impossible to write those lifetimes. In the Rust program, all explicit lifetime annotations correspond to any borrow that happened **outside** the function, and therefore, it is alive for the whole body of the function. Explicit lifetime annotations corresponding to regions spanning only a part of the function body would be pointless. Borrows inside a function can be analysed precisely by the borrow-checker. Explicit annotations are only used to represent constraints following from the code that the borrow-checker cannot see.

```
let mut x;
if (b) {
    x = a; // a: &'a T
} else {
    x = b; // b: &'b T
}
```

Example: We need to infer the type of `x`, such that it is a subtype of both `&'a T` and `&'b T`. We need to make sure that if we further use `x`, that is safe with regard to all loans that it can contain (here `a` or `b`).

In Rust, unlike in object-oriented languages like Java or C++, the only subtyping relation other than identity is caused by lifetimes¹⁰. Two regions (corresponding to lifetimes) can be either unrelated, a subset of each other (in terms of a set of CFG nodes) (denoted `'a: 'b`), or equal (typically a result of `'a: 'b` and `'b: 'a`). The dependency of subtyping on the inner parameter is called variance.

`F<T>` is covariant over `T` if `T` being a subtype of `U` implies that `F<T>` is a subtype of `F<U>` (subtyping “passes through”)

`F<T>` is contravariant over `T` if `T` being a subtype of `U` implies that `F<U>` is a subtype of `F`

`F<T>` is invariant over `T` otherwise (no subtyping relation can be derived)

[3]

Let us see what that means on an example specific to lifetimes. For a simple reference type `&'a T`, the lifetime parameter `'a` is covariant. That means that if we have a reference `&'a T` we can coerce it to `&'b T`, then `'a` is a subtype of `'b`. In

¹⁰During the type inference computation, there can also be a subtyping relation with a general kind of types (like `i32`), which is mostly used for not annotated literals, where we know it is “some kind” of integer, but we do not yet know which one

other words, if we are storing a reference to some memory, it is sound to assign it to a reference that lives for a shorter period of time. That is, if it is safe to dereference a reference within any point of period 'a, it is also safe to dereference it within any point of period 'b, ('b is a subset of 'a)¹¹.

The situation is different when we pass a reference to a function as an argument. In that case, the lifetime parameter is contravariant. For function parameters, we need to ensure that the parameter lives as long as the function needs it to. If we have a function pointer of type `fn foo<'a>(x: &'a T)`, we can coerce it to `fn foo<'b>(x: &'b T)`, where 'b lives longer than 'a. This conversion is safe because it only restricts the possible values of the parameter x.

Let us look at that visually. In the following code, we have region 'a, where it is safe to reference the storage of x, and region 'b where it is safe to reference the storage of y. If a function safely works with a reference of lifetime 'b, it will also safely work with a reference of lifetime 'a.

```
let x = 5;           // region 'a
{
    let y = 7;       //           // region 'b
}
```

The return type of the function is effectively an assignment to a local variable (just across function boundaries) and, therefore, is covariant.

The situation gets interesting when the two rules are combined. Let us have a function `fn foo<'a>(x: &'a T) -> &'a T`. The return type requires the function to be covariant over 'a, while the parameter requires it to be contravariant. This is called *invariance*.

For non-generic types, its variance immediately follows from the type definition. For generic types, the situation is more complex.

5.6.2 Variance of Generic Types

There are multiple approaches to the variance of generic types. It can be either derived from the usage of the type or its definition[12]. For non-generic types, use-site variance is used.¹² For generic types, Rust uses definition-site variance. That means that the variance is computed solely from the definition of the type (effectively, usage constraint to the body of the type), not from its usage (inside functions). The situation gets complicated when a generic type is used inside another generic type, possibly even in a recursive fashion. In that situation, the variance has to be computed using a fixed-point algorithm (further referred to as the “variance analysis”).

¹¹A subset of CFG nodes.

¹²For `&'a T`, if the reference is used as a function parameter, it is contravariant; if it is used as a return type, it is covariant.

5.6.2.1 Variance Analysis

Both `rustc` and `gccrs` variance analysis implementation is based on Section 4 of the paper [12]. Notation from the paper is followed in the documentation of both compilers, their documentation, and in this text. The paper primarily focuses on complex type variance, like in the case of Java, but it introduces an effective formal calculus, which works nicely with higher-kinded lifetimes.

The exact rules are best understood from the paper and from the code itself. Therefore, we will only provide a simple overview here. The analysis uses an iterative fixed-point computation, where variables form a semi-lattice with an additional binary operation. A single variable corresponds to a single lifetime or type parameter. Variables are initialized as bivariant.

The visitor traverses each type with the current variance of the visited expression as an input. Each member of a type is in a covariant position. Each member of a function parameter is in a contravariant position. The return type is in the covariant position. The generic argument position is determined by the variance of the generic parameter (a variable of this computation). The variance of the current node within the type is computed by the `transform` function, taking the variance of the parent node and the variance based on the position of the current node and building and expression. When a lifetime or type parameter is encountered, then if the current variance expression is constant, the variable is updated to the new variance using the join operation with the current value. For an expression containing at least one variable, the expression is added to the list of constraints. Here, the fixed-point computation requirement arises.

Once all types in the crate are processed, the constraints are solved using a fixed-point computation. Note that the current crate can use generic types from other crates, and therefore, it has to export/load the variance of public types.

Example of Algorithm Execution

```
struct Foo<'a, 'b, T> {
    x: &'a T,
    y: Bar<T>,
}
```

- Struct foo has three generic parameters, leading to 3 variables. $f0=o$, $f1=o$ and $f2=o$.
- x is processed first, in covariant position.
- $\&'a\ T$ is in covariant position, therefore variables are updated to $f0=+$ and $f2=+$.
- y is processed second, in covariant position.
- $\text{Bar}<T>$ is in covariant position.
 - T is inside a generic argument, therefore, its position is computed as a term $\text{transform}(+, b0)$.
 - * New constant $f2 = \text{join}(f2, \text{transform}(+, b0))$ is added.
- All types are processed. Let us assume that Bar is an external type with variances $[-]$ Now a fixed-point computation is performed.
- Iteration 1:
 - Current values are $f0=+$, $f1=o$ and $f2=+$
 - Processing constraint $f2 = \text{join}(f2, \text{transform}(+, b0))$
 - $\text{transform}(+, b0)$ where $b0=-$ yields $-$
 - $\text{join}(+, -)$ yields $*$
 - $f2$ is updated, therefore, another iteration is needed.
- Iteration 2:
 - Current values are $f0=+$, $f1=o$ and $f2=*$
 - Processing constraint $f2 = \text{join}(f2, \text{transform}(+, b0))$
 - $\text{transform}(+, b0)$ where $b0=-$ yields $-$
 - $\text{join}(*, -)$ yields $*$
 - $f2$ is not updated, therefore, the computation is finished.
- The final variance is $f0=+$, $f1=o$ and $f2=*$:
- $f0$ is evident,
- $f1$ stayed bivariant, because it was not mentioned in the type,
- $f2$ is invariant, because it is used in both covariant and contravariant position.

5.7 Error Reporting

As each function is analyzed separately, the compiler can easily report which functions violate the rules. Currently, only the kind of violation is communicated from the Polonius engine to the compiler. More detailed reporting is an issue for future work.

There are three possible ways the more detailed reporting could be implemented.

The first is to pass all the violations back to the compiler to be processed as a return value of the Polonius FFI invocation. This variant provides a simple separation of roles between the compiler and the analysis engine. However, it might be difficult

to implement correctly with regard to memory ownership around the FFI boundary since Polonius would need to allocate dynamically sized memory to pass the result. Polonius would need to implement a special API to release the memory.

The second variant is to pass a callback function for reporting the found errors to the Polonius engine, which would be called for each violation. However, Polonius only has information in terms of the enumerated nodes of the control flow graph. Therefore, a pointer to an instance of the borrow checker would need to be passed to the Polonius engine to be used in combination with the callback to resolve the nodes to the actual code. The separation of roles, where Polonius and Polonius FFI are used just as external computation engines, is broken.

A compromise between these variants would be to provide Polonius with callback functions, which would send the violations to the compiler one by one, leaving the allocation on the compiler side only.

Moreover, the borrow-checker currently does not store information to map the nodes back to source code locations. This issue is clearly technical only, and the functionality can be added easily with local changes only. Since this work has an experimental character, work on the analysis itself was prioritized over more detailed error reporting.

The final stage of the borrow-checker development would be to implement heuristics to guess the reason for the error and suggest possible fixes.

Chapter 6

Current State

TODO

This section describes the current state of the borrow checker. As explained in the introduction, this work was experimental, and it focused on exploring as many aspects of the problem as possible. Given the complexity of the Rust language and the borrow-checking itself, there are many parts of the borrow-borrow checker that have not been implemented or have been implemented using a temporary solution.

This project has demonstrated the feasibility of the selected approach and mapped the work necessary to create a production-ready solution. It has designed the infrastructure needed to solve the problem as the unimplemented parts are mostly technical.

6.1 Kind of Detected Errors

6.2 Parsing

Parsing handles both explicit and implicit lifetimes correctly.

Parsing of special lifetimes ('static and '_) was fixed. Handling of implicit lifetimes was added.

6.3 AST to HIR Lowering

6.4 Type Checking (TyTy Representation)

The resolution of named lifetimes to their binding clauses was added. TyTy types were refactored from the usage of named lifetimes to resolved regions. Previously, the handling of lifetimes in generic types was completely missing, and the representation of regions inside generic types was added. Also, a mechanism to map original types to substituted ones, preserving information about parameter position, was added.

6.5 Borrow-checker Scheduling

All top-level functions which are

6.6 BIR Building

The BIR builder can handle

The following constructs are not handled by the BIR builder and should be removed from HIR and desugared on the AST->HIR boundary: `ErrorPropagationExpr` (question mark operator), `WhileLetExpr`. Other constructs requiring pattern matching (`MatchExpr`, `IfLetExpr`, `IfLetConseqElse`) are not implemented. They require pattern validation to be implemented in BIR. Pattern destructuring is implemented except for the `AltPattern` (not supported by the rest of the compiler). Pattern destruction when no initial expression is provided is partly implemented (this construct is almost useless, and therefore, it has little priority).

6.6.1 BIR Dump

The BIR dump can display the BIR in a fashion very similar to MIR. It performs some transformation, like place renumbering, to produce a result closer to MIR. Those transformations can be easily turned off in the code. In addition to MIR, statement numbers within basic blocks is displayed to simplify debugging. All BIR constructs are supported.

Examples of BIR dump and MIR dump of the same program can be found in the appendix.

6.7 BIR Fact Collection

6.8 Polonius FFI

Interface with Polonius is realized using C ABI implemented in gccrs and a in a Rust crate “ffi-polonius”

6.9 Error Reporting

The borrow-checker report detected errors on the function level. It differentiates three kinds of errors: general loan errors, which are triggered by loan invalidations; subset errors, when insufficient constraints are specified; and move errors, following from possible use of uninitialized/deinitialized variables. Debug output can be used to link the error to BIR objects.

Further work is needed to transfer violated facts from Polonius to gccrs, translate them to source code locations, and figure out the reasons for those errors. Rustc also sorts the errors based on the reason priority to display only the useful ones to the user. For example, errors on temporary variables created by the compiler have very low priority.

Appendix A

Rustc Intermediate Representations Examples

Compilation commands:

```
$ rustc -Z unpretty=ast-tree  
$ rustc -Z unpretty=hir-tree  
$ rustc -Z unpretty=mir -Z identify-regions
```

A.1 Rust source code

```
struct Foo(i32);  
  
fn foo(x: i32) -> Foo {  
    Foo(x)  
}
```

A.2 Abstract Syntax Tree (AST)

```

Fn {
  defaultness: Final,
  generics: Generics {
    params: [],
    where_clause: WhereClause {
      has_where_token: false,
      predicates: [],
      span: simple.rs:3:22: 3:22 (#0),
    },
    span: simple.rs:3:7: 3:7 (#0),
  },
  sig: FnSig {
    header: FnHeader { unsafety: No, asyncness: No, constness: No },
    decl: FnDecl {
      inputs: [
        Param {
          attrs: [],
          ty: Ty {
            id: NodeId(4294967040),
            kind: Path(
              None,
              Path {
                span: simple.rs:3:11: 3:14 (#0),
                segments: [
                  PathSegment {
                    ident: i31#0,
                    id: NodeId(4294967040),
                    args: None,
                  },
                ],
                tokens: None,
              },
            ),
            span: simple.rs:3:11: 3:14 (#0),
            tokens: None,
          },
        ],
      ),
      pat: Pat {
        id: NodeId(4294967040),
        kind: Ident(
          BindingAnnotation(No, Not),
          x#0,
          None,
        ),
        span: simple.rs:3:8: 3:9 (#0),
        tokens: None,
      },
      id: NodeId(4294967040),
    },
  },
  span: simple.rs:3:7: 3:7 (#0),
  tokens: None,
}

```



```

        span: simple.rs:3:8: 3:14 (#0),
        is_placeholder: false,
    },
],
output: Ty(
    Ty {
        id: NodeId(4294967040),
        kind: Path(
            None,
            Path {
                span: simple.rs:3:19: 3:22 (#0),
                segments: [
                    PathSegment {
                        ident: Foo#0,
                        id: NodeId(4294967040),
                        args: None,
                    },
                ],
                tokens: None,
            },
        ),
        span: simple.rs:3:19: 3:22 (#0),
        tokens: None,
    },
),
span: simple.rs:3:1: 3:22 (#0),
},
body: Some(
    Block {
        stmts: [
            Stmt {
                id: NodeId(4294967040),
                kind: Expr(
                    Expr {
                        id: NodeId(4294967040),
                        kind: Call(
                            Expr {
                                id: NodeId(4294967040),
                                kind: Path(
                                    None,
                                    Path {
                                        span: simple.rs:4:5: 4:8 (#0),
                                        segments: [
                                            PathSegment {
                                                ident: Foo#0,
                                                id: NodeId(4294967040),
                                                args: None,
                                            },
                                        ],

```

```

],
  tokens: None,
},
),
span: simple.rs:4:5: 4:8 (#0),
attrs: [],
tokens: None,
},
[
  Expr {
    id: NodeId(4294967040),
    kind: Path(
      None,
      Path {
        span: simple.rs:4:9: 4:10 (#0),
        segments: [
          PathSegment {
            ident: x#0,
            id: NodeId(4294967040),
            args: None,
          },
        ],
        tokens: None,
      },
    ),
    span: simple.rs:4:9: 4:10 (#0),
    attrs: [],
    tokens: None,
  },
],
),
span: simple.rs:4:5: 4:11 (#0),
attrs: [],
tokens: None,
},
),
span: simple.rs:4:5: 4:11 (#0),
},
],
id: NodeId(4294967040),
rules: Default,
span: simple.rs:3:23: 5:2 (#0),
tokens: None,
could_be_bare_literal: false,
},
),
}

```

A.3 High-Level Intermediate Representation (HIR)

```

Fn(
  FnSig {
    header: FnHeader {
      unsafety: Normal,
      constness: NotConst,
      asyncness: NotAsync,
      abi: Rust,
    },
    decl: FnDecl {
      inputs: [
        Ty {
          hir_id: HirId(DefId(0:6 ~ simple[415f]::foo).10),
          kind: Path(
            Resolved(
              None,
              Path {
                span: simple.rs:3:11: 3:14 (#0),
                res: PrimTy(
                  Int(
                    I32,
                  ),
                ),
                segments: [
                  PathSegment {
                    ident: i32#0,
                    hir_id: HirId(
                      DefId(0:6 ~ simple[415f]::foo).11),
                    res: PrimTy(
                      Int(
                        I32,
                      ),
                    ),
                    args: None,
                    infer_args: false,
                  },
                ],
              },
            ),
          span: simple.rs:3:11: 3:14 (#0),
        },
      ],
      output: Return(
        Ty {
          hir_id: HirId(DefId(0:6 ~ simple[415f]::foo).12),
          kind: Path(
            Resolved(
              None,
              Path {
                span: simple.rs:3:19: 3:22 (#0),
                res: Def(
                  Struct,
                  DefId(0:3 ~ simple[415f]::Foo),
                ),
              },
            ),
          ),
        },
      ),
    },
  ),
)

```

```

    ),
    segments: [
        PathSegment {
            ident: Foo#0,
            hir_id: HirId(
                DefId(0:6 ~ simple[415f]::foo).13),
            res: Def(
                Struct,
                DefId(0:3 ~ simple[415f]::Foo),
            ),
            args: None,
            infer_args: false,
        },
    ],
},
),
),
span: simple.rs:3:19: 3:22 (#0),
},
),
c_variadic: false,
implicit_self: None,
lifetime_elision_allowed: false,
},
span: simple.rs:3:1: 3:22 (#0),
},
Generics {
    params: [],
    predicates: [],
    has_where_clause_predicates: false,
    where_clause_span: simple.rs:3:22: 3:22 (#0),
    span: simple.rs:3:7: 3:7 (#0),
},
BodyId {
    hir_id: HirId(DefId(0:6 ~ simple[415f]::foo).9),
},
)

...

Expr {
    hir_id: HirId(DefId(0:6 ~ simple[415f]::foo).3),
    kind: Call(
        Expr {
            hir_id: HirId(DefId(0:6 ~ simple[415f]::foo).4),
            kind: Path(
                Resolved(
                    None,
                    Path {
                        span: simple.rs:4:5: 4:8 (#0),
                        res: Def(
                            Ctor(
                                Struct,
                                Fn,
                            ),
                        ),
                    },
                ),
            ),
        },
    ),
}

```

```

        DefId(0:4 ~ simple[415f]::Foo::{constructor#0})),
    ),
    segments: [
        PathSegment {
            ident: Foo#0,
            hir_id: HirId(DefId(0:6 ~ simple[415f]::foo).5),
            res: Def(
                Ctor(
                    Struct,
                    Fn,
                ),
                DefId(0:4 ~ simple[415f]::Foo::{constructor#0})),
            ),
            args: None,
            infer_args: true,
        },
    ],
    },
    ),
    span: simple.rs:4:5: 4:8 (#0),
},
[
    Expr {
        hir_id: HirId(DefId(0:6 ~ simple[415f]::foo).6),
        kind: Path(
            Resolved(
                None,
                Path {
                    span: simple.rs:4:9: 4:10 (#0),
                    res: Local(
                        HirId(DefId(0:6 ~ simple[415f]::foo).2),
                    ),
                    segments: [
                        PathSegment {
                            ident: x#0,
                            hir_id: HirId(
                                DefId(0:6 ~ simple[415f]::foo).7),
                            res: Local(
                                HirId(
                                    DefId(0:6 ~ simple[415f]::foo).2),
                                ),
                            args: None,
                            infer_args: true,
                        },
                    ],
                },
            ),
        ),
        span: simple.rs:4:9: 4:10 (#0),
    },
],
),
span: simple.rs:4:5: 4:11 (#0),
}

```

A.4 Mid-Level Intermediate Representation (MIR)

```
fn foo(_1: i32) -> Foo {
    debug x => _1;
    let mut _0: Foo;

    bb0: {
        _0 = Foo(_1);
        return;
    }
}

fn Foo(_1: i32) -> Foo {
    let mut _0: Foo;

    bb0: {
        _0 = Foo(move _1);
        return;
    }
}
```

Appendix B

Comparison of BIR and MIR

BIR and MIR dump of the following code are displayed parallel, BIR on left pages and MIR on right pages. Note that assert macros in MIR were simplified to fit onto the page.

B.1 Compilation Commands

```
$ crab1 -frust-incomplete-and-experimental-compiler-do-not-use \  
        -frust-dump-bir -frust-borrowcheck  
$ rustc -Zdump-mir=nll -Zidentify-regions
```

B.2 Rust Source Code

```
pub fn fib(n: u32) -> u32 {  
    if n == 0 || n == 1 {  
        1  
    } else {  
        fib(n-1) + fib(n - 2)  
    }  
}
```

B.3 BIR (Rustc GCC)

```

fn fib(_2: u32) -> u32 {
    let _1: u32;    []
    let _2: u32;    []
    let _3: bool;   []
    let _5: u32;    []
    let _6: bool;   []
    let _8: u32;    []
    let _9: bool;   []
    scope 2 {
        let _14: u32; []
        let _15: u32; []
        let _16: u32; []
        let _19: u32; []
        let _20: u32; []
        let _21: u32; []
    }

    bb0: {
0    StorageLive(_3);
1    StorageLive(_5);
2    _5 = _2;
3    StorageLive(_6);
4    _6 = Operator(move _5, const u32);
5    switchInt(move _6) -> [bb1, bb2];
    }

    bb1: {
0    _3 = const bool;
1    goto -> bb3;
    }

    bb2: {
0    StorageLive(_8);
1    _8 = _2;
2    StorageLive(_9);
3    _9 = Operator(move _8, const u32);
4    _3 = move _9;
5    goto -> bb3;
    }

    bb3: {
0    switchInt(move _3) -> [bb4, bb5];
    }

    bb4: {
0    _1 = const u32;
1    goto -> bb8;
    }

    bb5: {
0    StorageLive(_14);

```


B.4 MIR (rustc)

```

fn fib(_1: u32) -> u32 {
    debug n => _1;
    let mut _0: u32;
    let mut _2: bool;
    let mut _3: u32;
    let mut _4: bool;
    let mut _5: u32;
    let mut _6: u32;
    let mut _7: u32;
    let mut _8: u32;
    let mut _9: (u32, bool);
    let mut _10: u32;
    let mut _11: u32;
    let mut _12: u32;
    let mut _13: (u32, bool);
    let mut _14: (u32, bool);

    bb0: {
        StorageLive(_2);
        StorageLive(_3);
        _3 = _1;
        _2 = Eq(move _3, const 0_u32);
        switchInt(move _2) -> [0: bb2, otherwise: bb1];
    }

    bb1: {
        StorageDead(_3);
        goto -> bb3;
    }

    bb2: {
        StorageDead(_3);
        StorageLive(_4);
        StorageLive(_5);
        _5 = _1;
        _4 = Eq(move _5, const 1_u32);
        switchInt(move _4) -> [0: bb4, otherwise: bb3];
    }

    bb3: {
        StorageDead(_5);
        _0 = const 1_u32;
        goto -> bb10;
    }

    bb4: {
        StorageDead(_5);
        StorageLive(_6);
        StorageLive(_7);
        StorageLive(_8);
        _8 = _1;

```

```

1  _14 = _2;
2  StorageLive(_15);
3  _15 = Operator(move _14, const u32);
4  StorageLive(_16);
5  _16 = Call(fib)(move _15) -> [bb6];
}

bb6: {
0  StorageLive(_19);
1  _19 = _2;
2  StorageLive(_20);
3  _20 = Operator(move _19, const u32);
4  StorageLive(_21);
5  _21 = Call(fib)(move _20) -> [bb7];
}

bb7: {
0  _1 = Operator(move _16, move _21);
1  StorageDead(_21);
2  StorageDead(_20);
3  StorageDead(_19);
4  StorageDead(_16);
5  StorageDead(_15);
6  StorageDead(_14);
7  goto -> bb8;
}

bb8: {
0  StorageDead(_9);
1  StorageDead(_8);
2  StorageDead(_6);
3  StorageDead(_5);
4  StorageDead(_3);
5  return;
}
}

```

B. Comparison of BIR and MIR

```
    _9 = CheckedSub(_8, const 1_u32);
    assert(!move (_9.1: bool)) -> [success: bb5, unwind: bb11];
}

bb5: {
    _7 = move (_9.0: u32);
    StorageDead(_8);
    _6 = fib(move _7) -> [return: bb6, unwind: bb11];
}

bb6: {
    StorageDead(_7);
    StorageLive(_10);
    StorageLive(_11);
    StorageLive(_12);
    _12 = _1;
    _13 = CheckedSub(_12, const 2_u32);
    assert(!move (_13.1: bool)) -> [success: bb7, unwind: bb11];
}

bb7: {
    _11 = move (_13.0: u32);
    StorageDead(_12);
    _10 = fib(move _11) -> [return: bb8, unwind: bb11];
}

bb8: {
    StorageDead(_11);
    _14 = CheckedAdd(_6, _10);
    assert(!move (_14.1: bool)) -> [success: bb9, unwind: bb11];
}

bb9: {
    _0 = move (_14.0: u32);
    StorageDead(_10);
    StorageDead(_6);
    goto -> bb10;
}

bb10: {
    StorageDead(_4);
    StorageDead(_2);
    return;
}

bb11 (cleanup): {
    resume;
}

}
```

Appendix C

Examples of Errors Detected by the Borrow-checker

A faulty program from gccrs test suite together with a fixed alternative (when applicable) is presented. Expected errors are marked using special comments used by the DejaGnu compiler testing framework.

C.1 Move Errors

A simple test, where an instance of type A, which is not trivially copiable (does not implement the compy trait) is moved twice.

```
fn test_move() {  
    // { dg-error "Found move errors in function test_move" }  
    struct A {  
        i: i32,  
    }  
    let a = A { i: 1 };  
    let b = a;  
    let c = a;  
}
```

```
fn test_move_fixed() {  
    let a = 1; // a is now primitive and can be copied  
    let b = a;  
    let c = b;  
}
```

More complex text test, where moves the occurrence of the error depends on runtime values. Error is raised because for some values, the violation is possible

```
fn test_move_conditional(b1: bool, b2:bool) {  
    // { dg-error "Found move errors in function test_move" }  
    struct A {  
        i: i32,  
    }  
  
    let a = A { i: 1 };  
    let b = a;  
    if b1 {  
        let b = a;  
    }  
    if b2 {  
        let c = a;  
    }  
}
```

```
fn test_move_fixed(b1: bool, b2:bool) {  
  
    let a = 1; // a is now primitive and can be copied  
    let b = a;  
    if b1 {  
        let b = a;  
    }  
    if b2 {  
        let c = a;  
    }  
}
```

C.2 Subset Errors

TODO

C.3 Loan Error

TODO

The following test were used when Polonius was first experimentally integrated into rustc.

In this test `s` is moved while it is borrowed. The test checks that facts are corectly propagated through the function call.

```
fn foo<'a, 'b>(p: &'b &'a mut usize) -> &'b&'a mut usize {
    p
}

fn well_formed_function_inputs() {
    // { dg-error "Found loan errors in function well_formed..."
    let s = &mut 1;
    let r = &mut *s;
    let tmp = foo(&r );
    s; //~ ERROR
    tmp;
}
```

This test check that variable cannot be used while borrowed.

```
pub fn use_while_mut() {
    // { dg-error "Found loan errors in function use_while_mut" }
    let mut x = 0;
    let y = &mut x;
    let z = x; //~ ERROR
    let w = y;
}
```

This test is similar to the previous one but uses a reborrow of a reference passed as an argument.

```
pub fn use_while_mut_fr(x: &mut i32) -> &mut i32 {
    // { dg-error "Found loan errors in function use_while_mut_fr" }
    let y = &mut *x;
    let z = x; //~ ERROR
    y
}
```

Appendix D

References

- [1] “*Software Memory Safety*” *Cybersecurity Information Sheet*. The National Security Agency, 2022.
- [2] Mitre, “CWE-416: Use after free.” <https://cwe.mitre.org/data/definitions/416.html>.
- [3] *The rust reference*. rust-lang.org, 2023.
- [4] N. Matsakis, “2094-nll,” in *The rust RFC book*, <https://rust-lang.github.io/rfcs/2094-nll.html>, 2017.
- [5] A. Stjerna, “Modelling rust’s reference ownership analysis declaratively in datalog,” Master’s thesis, Uppsala University, 2020. Available: <https://www.diva-portal.org/smash/get/diva2:1684081/FULLTEXT01.pdf>
- [6] “Polonius revisited, part 1 · baby steps — smallcultfollowing.com.” <https://smallcultfollowing.com/babysteps/blog/2023/09/22/polonius-part-1/>.
- [7] N. Matsakis, R. Rakic, *et al.*, *The polonius book*. rust-lang.org, 2021.
- [8] *Reference*, <https://llvm.org/docs/Reference.html>¹. LLVM Project, 2023.
- [9] R. M. Stallman and the GCC Developer Community, *GNU compiler collection internals*. Free Software Foundation, 2023.
- [10] *Rust Compiler Development Guide*. rust-lang.org; <https://rustc-dev-guide.rust-lang.org/index.html>.
- [11] <https://gcc-rust.zulipchat.com/#narrow/stream/281658-compiler-development/topic/Borrowchecking.20vs.20.28H.29IR>.
- [12] J. Altidor, S. S. Huang, and Y. Smaragdakis, “Taming the wildcards: Combining definition- and use-site variance,” *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 602–613, Jun. 2011, doi: 10.1145/1993316.1993569².