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.
Project reviewer: MSc. Arthur Cohen

Study programme: Open Informatics
Branch of study: Computer Engineering

January 2024

Thesis Supervisor
Ing. Pavel Píša Ph.D.
Department of Control Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague

Project Reviewer
MSc. Arthur Cohen
Rust GCC Maintainer responsible to the GCC Steering Committee
Embecosm

MASTER‘S THESIS ASSIGNMENT

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 bezpečného přístupu k 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.&nbsp;34, no.&nbsp;3, p.&nbsp;103–104. DOI&nbsp;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 Píša, 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 Píša, Ph.D.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		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

Acknowledgement

I express my gratitude to Jeremy Bennett for providing me with the opportunity to work on this project. I would also like to thank Arthur Cohen and Philip Herron, the maintainers of the Rust GCC project, for their consultations and reviews, and Pavel Píša for my introduction to the professional open-source developer community.
Furthermore, I would like to acknowledge our entire study group, Max Hollmann, Matěj Kafka, Vojtěch Štěpančík and Jáchym Herynek, for endless technical discussions and mental support.
Finally, I would like to thank my family for their continuous support.

Declaration

Abstract

This thesis presents the first attempt to implement a memory safety analysis, known as the borrow checker, within the Rust GCC compiler. It utilizes the Polonius engine, which was designed as the next-generation borrow checker for rustc. The text describes the design of this analysis, the necessary modifications of the compiler, and compares the internal representations between rustc and gccrs. This comparison highlights the challenges in adapting the rustc borrow checker design to gccrs. The thesis concludes with a discussion of the results and known limitations.

Keywords: compiler, Rust, borrow checker, static analysis, GCC, Polonius

Abstrakt

Tato práce představuje první pokus o realizaci analýzy paměťové bezpečnosti, nazývané borrow-checker, v překladači Rust GCC. Analýza využívá systém Polonius, který je vytvořen jako nová generace borrow-checkeru pro překladač rustc. Práce popisuje návrh analýzy, úprav překladače a porovnává vnitřní reprezentaci překladačů rustc a gccrs a poukazuje na problémy při adaptaci návrhu borrow-checkeru z překladače rustc na překladač gccrs. Práci uzavírá diskusí o výsledcích a známých omezeních.

Klíčová slova: překladač, Rust, borrow checker, statická analýza, GCC, Polonius

Contents

1 Introduction **1**

2 The Problem of Borrow Checking **2**
2.1 The Evolution of Borrow Checking in Rustc 4

3 Polonius Engine **7**
3.1 Polonius Facts 8

4 Comparison of Internal Representations **10**
4.1 GCC and LLVM 10
4.2 Rustc Representation 11
4.3 Rust GCC Representation 14

5 Rust GCC Borrow Checker Design **16**
5.1 Analysis of the Fact Collection Problem 16
5.2 The Borrow Checking Process 18
5.3 Representation of Lifetimes in TyTy 19
5.4 Borrow Checker IR Design 20
5.5 BIR Building 23
5.6 BIR Fact Collection and Checking 23
5.6.1 Subtyping and Variance 24
5.6.2 Variance of Generic Types 26
5.7 Error Reporting 27

6 Implementation **29**
6.1 Limitations 31
6.1.1 BIR and BIR Builder 31
6.1.2 Parsing, AST, HIR, TyTy 32
6.1.3 Fact Collection 32
6.1.4 Polonius FFI and Error Reporting 32
6.2 Building, Usage, and Debugging 32

7 Conclusion **35**

A References **36**

B Rustc Intermediate Representations Examples **38**
B.1 Rust Source Code 38
B.2 Abstract Syntax Tree (AST) 39
B.3 High-Level Intermediate Representation (HIR) 42
B.4 Mid-Level Intermediate Representation (MIR) 45

C Comparison of BIR and MIR **46**
C.1 Compilation Commands 46
C.2 Rust Source Code 46
C.3 BIR (Rustc GCC) 48
C.4 MIR (rustc) 49

D Examples of Errors Detected by the Borrow-Checker **52**
D.1 Move Errors 52
D.2 Subset Errors 53
D.3 Loan Error 53

E Glossary **55**

List of Figures

3.1 Illustration of steps performed by Polonius to detect errors. (Adapted from [8].) 8

4.1 LLVM IR CFG Example (generated by Compiler Explorer) 11
4.2 Comparison of compiler pipelines with a focus on internal representations 12

5.1 Placement of the borrow checker IR in the compilation pipeline . . . 19

Chapter 1
Introduction

Rust is a modern systems programming language that aims to provide memory safety without runtime overhead[1]. To achieve this goal, a Rust compiler has to perform a static analysis to ensure that the memory safety rules are not violated. This analysis is commonly called the *borrow checker*. The borrow checker is a complex analysis that has been evolving throughout the history of the Rust language and its reference implementation compiler, *rustc*. It evolved from a simple lexical analysis to a control-flow sensitive analysis, gradually providing a more precise validation. The experimental version of the rustc compiler uses a new analysis engine and algorithm called *Polonius*. The algorithm changes some fundamental views on the internal semantics of the analysis to allow more programs to be accepted and provides better error reporting for rejected programs[2]. The Polonius Working Group[1] is planning to replace the current rustc borrow checker with one based on Polonius in the Rust language edition 2024[3].

Rust GCC, also known as gccrs, is one of the emerging alternative Rust compilers. Unlike mrustc and rustc_codegen_gcc, gccrs aims to build a complete general-purpose Rust compiler independent of rustc. Gccrs aims to offer a rustc-compatible, drop-in replacement, capitalizing on the mature and diverse features of the GCC infrastructure. GCC (compared to LLVM) offers more target platforms and different optimizations and provides security plugins (originally designed for C) that could be used to find errors in *unsafe*[2] Rust code[4]. The goal of this thesis was to start the development of a Polonius-based borrow checker in gccrs.

The first chapter introduces the problem of borrow checking. It gives a brief overview of the development of the borrow checker in the rustc compiler, up to the Polonius project. 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 that lead 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 implementation, its development, the current state and the known missing features and limitations.

[1]https://rust-lang.github.io/compiler-team/working-groups/polonius/
[2]https://doc.rust-lang.org/reference/unsafety.html

Chapter 2
The Problem of Borrow Checking

This section introduces the concept of borrow checking and traces its development within the Rust programming language. It presents the simple lexical approach, followed by an explanation of a more advanced control-flow sensitive analysis and an introduction to the Polonius analysis engine, the latest approach to borrow checking in Rust. Since this work utilizes the Polonius engine, it is described in more detail in the following chapter.

Typical programming language implementations manage memory with dynamic storage duration in one of two ways[1]. Languages like C employ manual memory management, where programmers explicitly allocate and free memory, a method prone to errors[5]. In contrast, higher-level languages such as Java and Python use automatic memory management, where runtime garbage collectors handle memory management tasks.

Addressing the pitfalls of manual memory management, languages like C++ and Zig[2] have introduced tools for more implicit memory deallocation. In simple situations, these tools tie memory deallocation to the destruction of objects, utilizing concepts like RAII[3], smart-pointers[4], and defer statements[5]. Here, the key difference from stack allocation is that the ownership can be dynamically transferred between objects. In more complex situations, where multiple objects share memory and deallocation is tied to the last object's destruction, these languages opt-in for runtime solutions like reference counting[6].

Despite these improvements, two serious problems remain. First, programmers can incorrectly establish and maintain ownership binds, especially during dynamic ownership transfers between objects. This issue can very often occur when interfacing systems with differing memory management models[7]. Second, issue appears when the ownership is not transferred, but a copy of the pointer is used temporarily (this is called *borrowing* in Rust). The assumption that the owning object will exist for the whole time this copy is used is often wrong. This kind of mistake is called a *dangling pointer.*[6].

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

[2]https://ziglang.org/

[3]https://en.cppreference.com/w/cpp/language/raii

[4]https://en.cppreference.com/w/cpp/memory#Smart_pointers

[5]https://ziglang.org/documentation/master/#defer

[6]https://en.wikipedia.org/wiki/Reference_counting

[7]An interface between a C++ application with STL-based[8] memory management and the Qt GUI framework[9], 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. These methods can only be differentiated using their documentation.

Rust's memory safety strategy builds upon the RAII approach, but it introduces a built-in static analysis, known as the borrow checker, to prevent the above-mentioned memory errors. To make the analysis feasible, Rust allows only a conservative subset of memory-safe operations. Furthermore, Rust adds additional limitations to ensure that memory use is safe even during multithreaded execution. Because these restrictions are very strict and would severely limit the language, Rust allows certain restrictions to be bypassed in *unsafe* code blocks, placing the responsibility for maintaining safety invariants on the programmer.

The key idea behind Rust's approach is the strict differentiation between ownership transfers and borrowing, achieved through its type system. An ownership transfer, called "move" in Rust (and C++), binds owned unique resources to another object, detaching them from the current object. Rust simplifies this operation to a mere bitwise copy by restricting objects from storing a reference to itself. It also ensures the original object cannot be used after the move operation.

For borrows, Rust uses static analysis to ensure that any borrowed object is not deallocated during its use, requiring the borrowed object to *outlive* the borrow. This analysis is limited to individual functions to ensure analysis feasibility. Programmers are required to formally describe the relationships of borrows within function inputs and outputs using so-called *lifetimes*. Lifetimes are a special kind of type parameter that can be used to describe a part of program where concerned references must be valid. One can image a lifetime as an inference variables, for which the compiler has to find a valid value (a subset of the program). Lifetime annotations are related using *outlives* relationships, indicating that one reference's lifetime is a subset of another.

Throughout Rust's borrow checking development, the interpretation of *a subset of the program* has evolved. Initially it was based on expressions, then control flow graph (CFG) points, and later potentially used borrows.

Example: Consider a vector-like structure storing references to integers without owning them. We introduce a lifetime parameter `'a` to represent the parts of the program where the vector must be valid, substituted with a concrete lifetime at each use site.

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

The add method has a separate lifetime parameter `'b` for the inserted reference. Each method invocation substitutes `'b` with the concrete lifetime of the reference. The compiler ensures `'b` outlives `'a` (imposed by the `'b: 'a` constraint), ensuring all references in the vector remain valid as long as the vector exists.

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

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[7], which introduced non-lexical borrow checking to Rust. Examples from the 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 expressions) 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 must be valid until the end of the scope of 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!  //   |
} // <------------------------------+
```

There is no simple way to determine (from the syntactic structure) when the lifetime of the reference should end to prove that his program is safe. This code can be fixed by explicitly specifying where the lifetime should end. However, this clutters the code and 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 occurs when lifetimes are not symmetric in conditional branches. A typical case is where a condition checks the presence of a value. In the positive branch, we have a reference to a value which is a part of the `map`, but in the negative branch, we do not. Therefore, it is safe to create a new reference in the negative branch. By *safe*, we mean that there will be only one reference pointing to the `map` object at any time. A convenient way to describe *at any time* is to use the control flow graph (CFG) 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 RFC 2094[7]. 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 of lifetimes as regions (set of points in CFG) where the reference is valid. The goal of the analysis is to find the smallest regions so 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. This approach is called NLL (non-lexical lifetimes) in rustc.

The next generation of borrow checker in Rust is based on the Polonius analysis engine. Polonius is an extension of NLL, which is capable of proving more programs to be safe by using a different interpretation of lifetimes.

Unlike NLL, Polonius can handle the last example. In this scenario the problem 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. This 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. Here 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 removes 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 Matsakis[1] and extended by Rémy Rakic[2] and Albin Stjerna[8] 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 by different research projects, making it suitable for usage in gccrs. Polonius interfaces with the compiler by passing around a struct of vectors[3] of facts, where each fact is represented by a tuple of integers[4] (or types convertible to integers). It is completely unaware of the compiler internals.

In the previous chapter, we 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 that it is the task of the analysis to compute it.

Polonius begins by processing the input facts, computing transitive closures of relationships and analyzing variable initializations and deinitializations across the CFG. It then proceeds to identify move errors, where the ownership of an object is erroneously transferred multiple times. In the next step, it calculates the liveness of variables and the "outlives" graph (transitive constraints of lifetimes at each CFG point)[9]. All origins that appear in the type of live variable are considered live.

The engine next determines *active loans* based on two criteria: the liveliness of any origin containing the loan (i.e., there is a variable that might reference it) and the fact variable/place referencing the loan was not reassigned.

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

[1]https://github.com/nikomatsakis

[2]https://github.com/lqd/

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

[4]`usize`

such a situation can happen, and if so, 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.

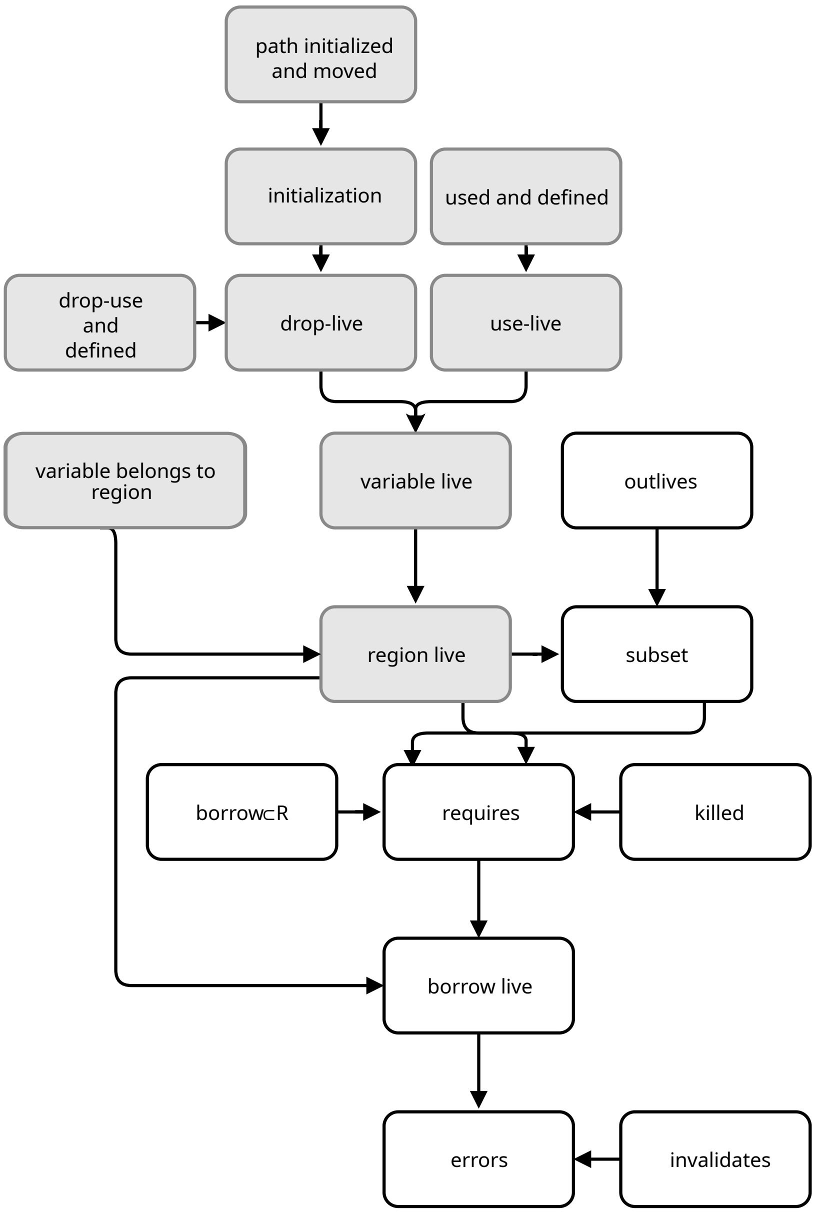


Figure 3.1: Illustration of steps performed by Polonius to detect errors. (Adapted from [8].)

3.1 Polonius Facts

This section outlines the facts that Polonius utilizes, offering a better idea of the work that the compiler needs to do. These facts are categorized and briefly explained. For an exhaustive list, refer to the Polonius source code[5] and the Polonius Book[10].

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

- Atoms:
  - `Point` is a CFG point.
  - `Variable` is a variable in the program.
  - `Path` is a memory location in the program.
  - `Origin` is a set of loans that can be referenced using a variable at each CFG point. It is the interpretation of lifetimes used by Polonius.
  - `Loan` is a result of a borrow expression.
- Control Flow Graph:
  - `cfg_edge: (Point, Point)` represent the edges in the control flow graph of the program.
- Variable Usage and Effects:
  - `var_used_at: (Variable, Point)` marks locations a variable is used in any way except for being dropped (destructed).
  - `var_defined_at: (Variable, Point)` marks the beginning of a variable's scope or its reassignment. All facts related to given variable are reset at this point.
  - `var_dropped_at: (Variable, Point)` indicates a point where a variable is dropped (its destructor is called).
  - `use_of_var_derefs_origin: (Variable, Origin)` means that a variable type contains given origin.
  - `drop_of_var_derefs_origin: (Variable, Origin)` reflects that the origin is used in the drop implementation.
- Path Usage and Effects: Paths correspond to indirect or partial access to a variable, such as field access or casting.
  - `path_is_var: (Path, Variable)` lists trivial paths that directly correspond to a variable.
  - `child_path: (Path, Path)` describes hierarchical relationships between paths, where one path is a subset or component of another.
  - `path_assigned_at_base: (Path, Point)` highlights where a specific path is assigned in the CFG.
  - `path_moved_at_base: (Path, Point)` marks the transfer of ownership of origins at a specific CFG point.
  - `path_accessed_at_base: (Path, Point)` indicates any memory access (read or write) to a path.
- Origin Relationships:
  - `known_placeholder_subset: (Origin, Origin)` constrains universal origins, representing loans from outside the function.
  - `universal_region: (Origin)` lists universal origins.
  - `subset_base: (Origin, Origin)` describes origin subset (outlives) relationships.
  - `placeholder: (Origin, Loan)` associates a universal origin with a loan that happened outside the function.
- Loan Facts:
  - `loan_issued_at: (Origin, Loan, Point)` marks execution of a borrow expression.
  - `loan_killed_at: (Loan, Point)` marks the end of a loan's validity.
  - `loan_invalidated_at: (Point, Loan)` marks points where an active loan leads to an error.

Chapter 4
Comparison of Internal Representations

Executing a borrow checker with an external analysis engine involves two key steps. The first is collecting relevant program information, referred to as *facts*. The second step is evaluation of these facts using the external engine. Before we can discuss the *collection* of facts, we need a clear understanding of 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 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 on which they are built (GCC and LLVM). We will only focus on the middle-end of each platform, since the back-end does not directly influence the front-end development.

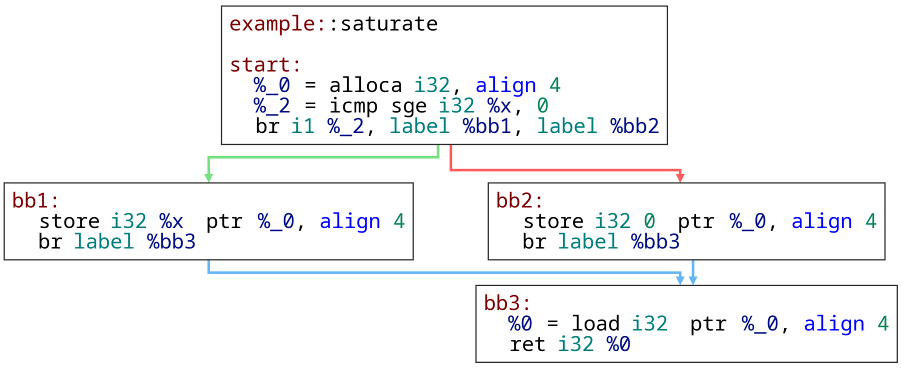
LLVM is built around a three-address code (3-AD)[1] representation known as the *LLVM intermediate representation* (LLVM IR)[11]. This IR serves as an interface between the front-ends and the LLVM platform. Each front-end is responsible for transforming its custom AST IR[2] into the LLVM IR. The LLVM IR is stable and strictly separated from the front-end; therefore, it cannot be easily extended to include language-specific constructs.

GCC, in contrast, uses a tree-based representation called GENERIC[12, p. 175] for interfacing with front-ends. 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.[12, p. 212] The GENERIC representation is subsequently transformed into GIMPLE, which is mostly[3] a 3-AD representation. This transformation involves decomposing expressions into a series of statements and introducing temporary variables. This

[1]Three-address code represents a program as sequences of statements (known as *basic blocks*), connected by control flow instructions to form a control flow graph (CFG).

[2]The abstract syntax tree (AST) is a structure representing the program's syntax. It is the direct result of parsing. For instance, an expression `1 + (2 - 7)` would be represented as a `subtraction` node, with children representing `1` and the subexpression `(2 - 7)`.

[3]"GIMPLE 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 such as lexical scopes and nested expressions, while "Low GIMPLE" exposes all of the implicit jumps for control and exception expressions directly in the IL and EH region trees."[12, p. 225]



```
example::saturate

start:
  %_0 = alloca i32, align 4
  %_2 = icmp sge i32 %x, 0
  br i1 %_2, label %bb1, label %bb2
```

```
bb1:
  store i32 %x  ptr %_0, align 4
  br label %bb3
```

```
bb2:
  store i32 0  ptr %_0, align 4
  br label %bb3
```

```
bb3:
  %0 = load i32  ptr %_0, align 4
  ret i32 %0
```

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

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.[12, 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 that 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 IR is language independent.

4.2 Rustc 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) [13] [4] [5]. As the AST is lowered to HIR, some complex language constructs are desugared to simpler constructs. For example, various types of loops are transformed into a single infinite loop construct (Rust `loop` keyword), and many structures that can perform pattern matching (`if let`, `while let`, `?` operator) are transformed into the 'match" construct[14] [6].

[4]https://rustc-dev-guide.rust-lang.org/macro-expansion.html
[5]https://rustc-dev-uide.rust-lang.org/name-resolution.html
[6]https://doc.rust-lang.org/reference/expressions/if-expr.html#if-let-expressions

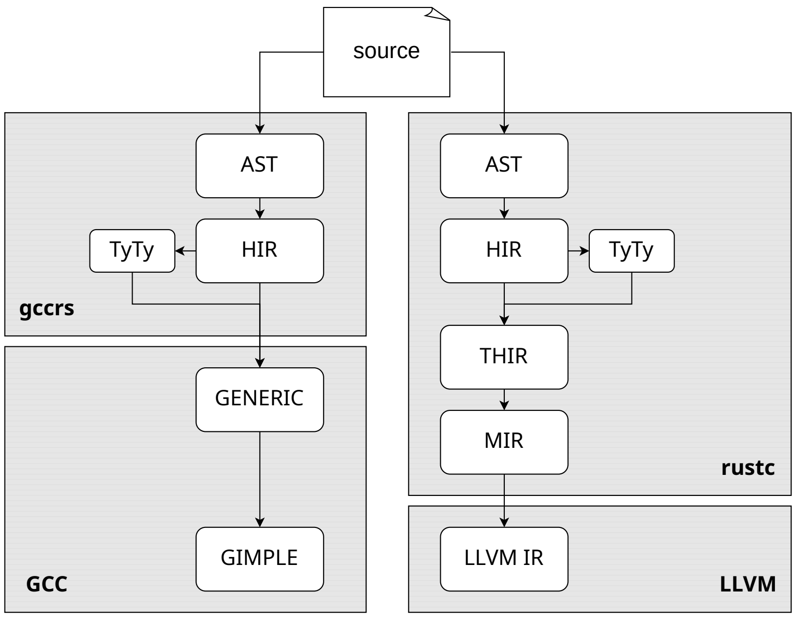


Figure 4.2: Comparison of compiler pipelines with a focus on internal representations

```
struct Foo(i31);

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

Example: This simple code snippet will serve as our example through this section.

```
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 } ]
        }
      },
  },
  ...
}
```

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.

The HIR is rustc primary representation, and it is used for most operations[13], HIR It combines a simplified AST with additional tables for quick access to additional information, such as expression and statement types. These tables are used for analysis passes, including full name resolution and type checking. Type checking includes verification type correctness, inference, and resolving of implicit type-dependent constructs[13] [7].

```
#[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 pattern[8]. 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. This 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

[7]https://rustc-dev-guide.rust-lang.org/type-checking.html
[8]https://en.wikipedia.org/wiki/Builder_pattern

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, or a cast).

Rustc also uses a special IR, called 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 HIR analysis. Each type (all its occurrences in the program) is represented by a single TyTy object instance. This is achieved by interning[9]. Note that there can be multiple equivalent types of different structures. 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 [13].

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 the rustc 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.

[9]https://en.wikipedia.org/wiki/Interning_%28computer_science%29

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 a 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 are 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 much more complex than the current approach. Given the limited development resources of gccrs, this is not a viable option.[15]

The 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 that 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 is 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. The options considered 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 sec. 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 checking[1]. 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.

The only IR in GCC that contains CFG information is GIMPLE; however, under

[1]The 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.

normal circumstances, GIMPLE is language agnostic. It is possible to annotate GIMPLE statements with language-specific information using special statements that 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 (which includes 11 passes), after which facts could be collected. After that, the facts would need to be discarded to avoid complicating the tens of subsequent passes[2][12, 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 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, such as 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 the 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 code already compiled. This fact would complicate the use of 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 performed 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 implemented regardless of the path taken by the borrow checker. Those transformations

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

include mainly loops and pattern-matching structures. These transformations are even documented in the rust reference[14].

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

To ensure that all possible approaches were considered, we 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 MIRI[3] with gccrs. The borrow checking would be very similar to rustc borrow checking, and parts of rustc 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 the 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[15] 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: building 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 continue. This IR, later called the borrow checker IR (BIR), focuses only on the flow of data, and 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 MIR as possible. This feature allows us to check the generated BIR against the MIR generated by rustc, at least for simple programs. The use of 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 the HIR is type-checked and TyTy types are created. The TyTy needs to resolve and store information about lifetimes and their corresponding 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 function must be memory-safe for every possible lifetime). In function defini-

[3]https://github.com/rust-lang/miri

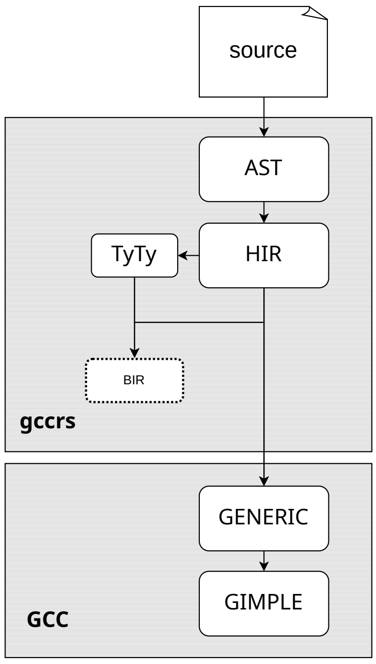


Figure 5.1: Placement of the borrow checker IR in the compilation pipeline

tions, lifetimes may be omitted (elided) if all references share 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. Subsequently, the collected facts are passed to Polonius, which then computes the results of the analysis. The results are then passed back to the compiler, which translates them into error messages.

5.3 Representation of Lifetimes in TyTy

In this work, the term lifetime refers 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[14][b], and local/existential 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 both this text and the implementation, the terms are used interchangeably.

[a]There are two kinds of lifetimes in Rust semantics: universal and existential. Universal lifetimes correspond to code that occurs outside the function. It is called universal because the concerned borrow checking rules use the universal quantifier. That means

[4]At 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.

that the function has to be valid *for all* possible outside code that satisfies the specified (or implied) constraints. Existential lifetimes correspond to the 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

In order to analyze more complex lifetimes than just simple references, it was necessary to add a 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 binding clauses. Gccrs recognizes four kinds of regions. In a function body, explicit lifetime annotations result in *named* lifetimes, while implicit annotations lead to *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` clause[5]. These 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 is 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. An issue arising from this automatic propagation is the updating of the bindings of early bound lifetimes. This means that by a simple inspection of the body of the generic type, one would not be able to resolve the lifetimes. A trick solves this problem. Each type in TyTy is identified by a unique 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 as 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. When necessary, the original ID can be used to look up the initial type.[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 Intermediate Representation (BIR) is a three-address code representation, designed to closely resemble a subset of rustc Mid-level Intermediate Representation (MIR). Like MIR, it represents the body of a single function (or function-like item, such as a closure), as borrow checking is performed on each function separately. It abstracts specific operations into a few key operations that focus on data flow.

[5]`for<'a> fn(&'a i32) -> &'a i32`

[6]This was once revealed to me in a dream.

```
fn fib(_2: u32) -> 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];
    }

   // ... (omitted for brevity)

    bb5: {
    0    StorageLive(_14);
    1    _14 = _2;
    2    StorageLive(_15);
    3    _15 = Operator(move _14, const u32);
    4    StorageLive(_16);
    5    _16 = Call(fib)(move _15) -> [bb6];
    }

    // ... (omitted for brevity)

    bb8: {
    0    StorageDead(_9);
    // ... (omitted for brevity)
    4    StorageDead(_3);
    5    return;
    }
}
```

Example: The following example shows a shortened BIR dump of a simple Rust program computing the nth Fibonacci number. The complete source code and full dump are available in appendix C.

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 keeps constants separate. The `Operand` enum is a common interface for places and constants. However, since operations use constants and lvalues in 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, because the number of fields is static and typically small, the size of arrays is unbound and depends on runtime information.

Structure of the BIR Function

- basic block list
  - basic block
  - `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

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

5.5 BIR Building

BIR construction involves visiting the High-Level Intermediate Representation (HIR) tree of the function. There are specialized visitors for expressions, statements, and patterns, as well as a top-level visitor for handling function headers. Whenever a new place is created in the compilation database, a corresponding list of fresh regions[8] is generated. Counting the number of lifetimes to be generated involves traversing the type structure. For generic types, the inner structure is ignored and only the lifetime and type parameters are considered. Note that the type parameters can be also generic, creating a structure known as higher-kinded[9]. All types are independently queried for each node from the HIR, instead of being derived within 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.

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 process extracts Polonius facts from the BIR and conducts additional checks. Polonius is responsible for verifying lifetime (region) constraints, ensuring each place is moved at most once, and checking that illegal accesses are not made to borrow memory locations. The collection process involves two phases: gathering static facts from the place database and universal region constraints, and traversing the BIR along the CFG to collect dynamic facts.

The fact collection is performed in two phases. First, static facts are collected from the place database. These include universal region constraints (constraints corresponding to lifetime parameters of the function) collected during BIR construction

[8]In 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

and facts collected from the place database. Polonius needs to know which places correspond to variables and which form paths (see the 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. The relations of the regions 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[10].

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 necessary to model the semantic aspects where the statement's effect is immediate or follows the execution of the statement. For each statement and (if present) its expression, Polonius facts are collected. These 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. Lifetimes not explicitly mentioned are inferred in the same way as type parts (e.g., `let a = (_, i32) = (true, 5);` infers the type to `(bool, i32)`). In Rust, explicit lifetime annotations in a function correspond to borrows that occurred *outside* the function, implying that these lifetimes span the entire function body. Annotations for lifetimes that cover only part of the function body would be redundant, as borrows within a function are precisely analyzed by the borrow checker. Explicit annotations are used only to represent constraints from code outside the function's scope.

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

Example: The type of `x` must be inferred to be a subtype of both `&'a T` and `&'b T`, ensuring safe use with all potential loans (here `a` or `b`).

In Rust, unlike object-oriented languages like Java or C++, the only subtyping relationship, apart from identity, arises from lifetimes[10]. Two regions (representing

[10]During type inference computation, there can also be subtyping relations with general kinds of

lifetimes) can either be unrelated, subsets of each other in terms of loans or CFG points (`'a: 'b`), or equal (resulting from `'a: 'b` and `'b: 'a`). The dependency of subtyping on the inner parameter is called variance.

Definition [14]
`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)

Consider an example specific to lifetimes in Rust. With a simple reference type `&'a T`, the lifetime parameter `'a` is covariant. This implies that a reference `&'a T` can be safely coerced into `&'b T` if `'a` is a subtype of `'b`. In practical terms, if it's safe to dereference a reference at any point during the period `'a`, it remains safe throughout the shorter period `'b`, given `'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. For instance, a function pointer with the type `fn foo<'a>(x: &'a T)` can be coerced into `fn foo<'b>(x: &'b T)` if `'b: 'a`. Such a transformation is safe as it narrows the range of acceptable argument values for the parameter `x`.

To visualize this concept, consider the following code snippet, where `'a` denotes a region safe for referencing the storage of `x`, and `'b` denotes a similar region for `y`. A function that operates correctly with a reference of lifetime `'b` is also guaranteed to work correctly with a reference of lifetime `'a`, since `'a` contains `'b`.

```
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 becomes 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, variance is directly derived from the type definition. However, variance in generic types is more complex and subject to different approaches.

types (like), which is mostly used for literals without a type annotation, where we know it is "some kind" of integer, but we do not yet know which one.

[11] A subset of CFG nodes.

5.6.2 Variance of Generic Types

Generic type variance can be derived from either the type's usage or its definition[16]. Rustc employs definition-site variance for generic types, meaning variance is computed from the type's definition, rather than its usage in functions. The situation becomes complicated when a generic type is used within another type, possibly in a recursive manner. In such cases, variance requires computation via a fixed-point algorithm, referred to as "variance analysis".

5.6.2.1 Variance Analysis

Both rustc and gccrs implement variance analysis based on section 4 of the paper [16]. The notation from the paper is followed in the documentation of both compilers, as well as in this text. While the paper primarily focuses on the variance of complex types, like in the case of Java, it introduces an effective formal calculus, which is also applicable to higher-kinded lifetimes.

For a thorough understanding of the exact rules, the paper and the source code are the best resources. Here, we provide only a basic overview. The analysis employs an iterative fixed-point computation, where variables form a semi-lattice with an additional binary operation. Each variable corresponds to a single lifetime or type parameter and is initially set as bivariant.

The visitor algorithm traverses each type, taking the current variance of the visited expression as input. Every type member is in a covariant position. Conversely, each function parameter member is in a contravariant position, while the return type is in a covariant position. The position of a generic argument is determined by the variance of the generic parameter (represented as a variable in this computation). The variance of the current node within the type is computed by a `transform` function, which considers both the parent node's variance and the current node's positional variance. 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 expressions containing at least one variable, the expression is added to a list of constraints and the fixed-point computation is used.

Example of Algorithm Execution

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

- `Foo` has three generic parameters, resulting in 3 variables: `f0=o`, `f1=o`, `f2=o`.
- `x` is first processed in the covariant position.
    - `&'a T` being in the covariant position updates the variables to `f0=+` and `f2=+`.
- `y` is next, also in the covariant position.
    - `Bar<T>` being in the covariant position.
        - `T` inside a generic argument leads to `transform(+, b0)` for

its position.
        - A new constant `f2 = join(f2, transform(+, b0))` is added.
- After processing all types and assuming `Bar` is an external type with variances `[-]`, a fixed-point computation begins.
    - Iteration 1:
        - Starting values: `f0=+`, `f1=o`, `f2=+`.
        - Processing constraint `f2 = join(f2, transform(+, b0))`.
        - `transform(+, b0)` with `b0=-` gives `-`.
        - `join(+, -)` results in `*`.
        - Update of `f2` requires another iteration.
    - Iteration 2:
        - Current values: `f0=+`, `f1=o`, `f2=*`.
        - Processing same constraint.
        - `transform(+, b0)` still yields `-`.
        - `join(*, -)` remains `*`.
        - No update to `f2`, computation concludes.
- Final variances: `f0=+`, `f1=o`, `f2=*`:
    - `f0` is evident.
    - `f1` remains bivariant, as it is unmentioned in the type.
    - `f2` is invariant due to its usage in both covariant and contravariant positions.

After processing all types in the crate, constraints are resolved using the fixed-point computation. Note that current crates might use generic types from other crates, necessitating the export/load of variance for public types.

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 area for future work.

There are three possible approaches for implementing more detailed reporting:

1. *Returning All Violations:* This method involves passing all violations back to the compiler as a return value of the Polonius FFI invocation. It offers a clear separation of roles between the compiler and the analysis engine. However, implementing this correctly could be challenging due to memory ownership concerns at the FFI boundary. Polonius would need to allocate dynamically sized memory for the result and provide an API for its release.

2. *Callback Function for Error Reporting:* Another option is to provide the Polonius engine with a callback function to report each found error. But, as Polonius only possesses information in terms of the enumerated nodes of the control flow graph, a pointer to an instance of the borrow checker would also need to be passed. This pointer would be used in conjunction with the callback to map nodes back to the actual code. However, this approach compromises

the separation of roles, where Polonius and Polonius FFI act solely as external computation engines.

3. *Compiler-Side Allocation via Callback Functions:* A compromise between these two methods would be to supply Polonius with callback functions that relay violations to the compiler one at a time, keeping memory allocation on the compiler side.

Additionally, the borrow checker does not currently store information to trace the nodes back to their source code locations. This limitation is purely technical and can be addressed straightforwardly with localized changes. Given the experimental nature of this work, the focus has been on analysis over detailed error reporting.

The final stage in developing the borrow checker would involve implementing heuristics to infer the reasons for errors and suggest potential fixes.

Chapter 6
Implementation

After the initial experiments described in sec. 5.1, the project was implemented in the following phases: First, an initial version of the borrow checker IR (BIR), lowering from HIR to BIR (the BIR builder), and a textual BIR dump were implemented. Second, the first version of the BIR fact collection and the Polonius FFI were added. At this stage, the first simple error detections were tested. Next, the implementation was extended to handle more complex data types, especially generics. Finally, the BIR fact collection was extended to handle the new information and emit all available facts.

The initial version of the borrow checker included only the minimal information that the borrow checker was expected to need. The builder was able to lower most operator and initializer expressions, borrow expressions, function calls, and simple control flow operations (`if`, `if/else`, `while`, `loop`, `return`). The compiler was extended[1] to handle labeled blocks[2] to lower (and test) `break` and `continue` expressions. Note that in the Rust language, `break` and `continue` can use label identifiers to exit nested loops or return a value from any labeled block[14].

The BIR dump was designed to be as similar to MIR as possible for manual verification. However, rustc performs many transformations on MIR, and there are various versions of the dump available. Originally, the MIR dump from the online Compiler Explorer[3] was used, but this version is optimized and cleaned up. It proved complicated to align with this dump, requiring additional BIR transformations. This led to the decision to change the reference MIR dump. MIR after each MIR pass can be exported from rustc using the `-Zdump-mir=*` flag. Additionally, the `-Zunpretty=mir` option is available. The logical choice was to use the MIR version for borrow checking (`-Zdump-mir=nll`), which is less optimized and contains additional borrow checking information. Most BIR transformations were removed after this change ot the reference MIR dump.

This initial BIR version and related infrastructure were submitted to Rust GCC in pull request 2702[4], adding 3,779 lines of new code, including a document titled “BIR Design Notes” to assist new developers with the borrow checker implementation. It is located in `gcc/rust/checks/errors/borrowck/bir-design-notes.md`[5].

In the second phase, the fact collection and an interface to the Polonius engine were implemented. Initially, only lifetimes of simple references were handled (at most one lifetime per type). Fact collection processed all places in the place database and

[1]https://github.com/Rust-GCC/gccrs/pull/2689
[2]https://doc.rust-lang.org/reference/expressions/loop-expr.html#labelled-block-expressions
[3]https://godbolt.org/
[4]https://github.com/Rust-GCC/gccrs/pull/2702
[5]https://github.com/Rust-GCC/gccrs/blob/df5b6a371dba385e4bb03ebd638cd473c4cc38eb/gcc/rust/checks/errors/borrowck/bir-design-notes.md

traversed the BIR control flow graphs. The interface to Polonius consists of a C ABI in gccrs and a C ABI (generated by rust-bindgen[6] and manually cleaned up and extended) in a small static Rust library (FFI Polonius). The FFI Polonius library's role is to invoke the Polonius engine. A discussion about integrating this interface into the GCC build system began in pull request draft 2716[7]. This integration is complex, requiring compilation of Rustc code beyond gccrs's current capabilities. For development purposes, the Cargo build system (rustc) is invoked from the GCC Makefile. While not ideal for production due to no cross-compilation handling, this solution is optimal for development. It was decided to keep the build integration downstream for the time being. The most viable solution for upstreaming is to release the Polonius FFI as a dynamic library, with the building process outside GCC. The final decision on this will be made when the borrow checker is ready for public release. Therefore, this phase was not submitted to Rust GCC, with newer independent commits rebased below the FFI commit and submitted separately. At this stage, the borrow checker successfully detected repeated moves[8], basic subset errors (i.e., insufficient constraints between inputs and outputs of functions), and moves behind a reference[9]. Error output was implemented using only the FFI Polonius debug output at this stage (see the example).

```
[34/35] Checking function test_move
Polonius analysis completed. Results:
Errors: {}
Subset error: {}
Move error: {
    GccrsAtom(
        11,
    ): [
        GccrsAtom(
            2,
        ),
    ],
}
```

Example: FFI Polonius debug output for a simple program with a move error. The output reports that at a CFG point encoded as the number 11, a path number 2 was moved illegally.

In the third (and final) phase, the entire borrow checker, the TyTy IR, and the type checker were extended to support complex types containing multiple regions (lifetimes). Variance analysis and helper region tools were implemented. The BIR builder and fact collection were expanded to handle the new information and emit all available facts. Correctly collecting facts is challenging due to limited documentation of the facts and their relationship with Rust code. The current implementation relies on the Polonius Book[10], the Polonius source code[17], the rustc source code[18], and experiments using rustc and the Polonius CLI. Some facts might be missing or incorrectly collected.

[6]https://github.com/rust-lang/rust-bindgen
[7]https://github.com/Rust-GCC/gccrs/pull/2716
[8]https://doc.rust-lang.org/error_codes/E0382.html
[9]https://doc.rust-lang.org/error_codes/E0507.html

The borrow checker could identify most errors that violate access rules (number of loans of a given type allowed, loan/access conflicts), move/initialization errors, and subset errors. To demonstrate its functionality, a small test suite was created based on tests from the Polonius project, supplemented with custom ones. Ideally, the borrow checker would be tested against rustc test suite, but gccrs is currently unable to compile most of these tests as they rely on the Rust standard library. Examples from the gccrs borrow checker test suite are available in the appendix.

This phase is pending final cleanup and submission in the branch borrowck-stage2[10]. It includes 5146 additions and 720 deletions and is expected to be submitted to Rust GCC soon.

6.1 Limitations

The main bottleneck in the current implementation is the BIR builder. After covering a subset of Rust sufficient for testing error detection capabilities, the focus shifted to other aspects of the borrow checker to implement all necessary parts, even if in a limited fashion. Below is a list of known limitations of the current implementation.

6.1.1 BIR and BIR Builder

- Currently, only nongeneric functions are supported (not closures or associated functions and methods[11]). Other function-like items require special top-level handling, though their body handling is identical. Generic functions must be monomorphised before checking.
- Method calls are not handled due to required implicit coercion of the `self` argument.
- The `?` operator and `while let` are not addressed. They will be removed at the `AST->HIR` boundary.
- Handling for `if let` and `match` expressions is missing, particularly for pattern detection (variant selection). Pattern destructuring is mostly implemented for `let` expressions and function parameters. The `or` pattern[12] is unsupported, as is pattern declaration without an initial value, except for identifier patterns[13].
- Enums[14] are not supported.
- Unsafe blocks are not handled.
- Asynchronous code is completely unsupported in the compiler.
- Unwind paths (drops) are not created, as drops are not supported by the compiler.
- Two-phase borrowing[15] is not implemented. While not essential for correctness, it reduces false positives.
- Location information is not stored, which is necessary for practical error reporting.

[10]https://github.com/jdupak/gccrs/tree/borrowck-stage2
[11]https://doc.rust-lang.org/nightly/reference/items/associated-items.html#associated-functions-and-methods
[12]https://doc.rust-lang.org/reference/patterns.html#or-patterns
[13]https://doc.rust-lang.org/reference/patterns.html#identifier-patterns
[14]https://doc.rust-lang.org/reference/types/enum.html
[15]https://rustc-dev-guide.rust-lang.org/borrow_check/two_phase_borrows.html

- Copy trait probing is not performed. The `Copy` trait is derived only for primitive types and tuples of primitive types.
- Not all fake operations (e.g., `fake_unwind`) are represented or emitted.
- Advanced projections like `cast` might require more complex handling.

6.1.2 Parsing, AST, HIR, TyTy

- Lifetime elision[16] is not handled.
- Variance analysis does not import or export variance information via metadata export and currently only considers one crate.
- Region propagation in the type checker requires further testing, particularly in cases involving traits.
- Late-bound lifetime[17] instantiation is unaddressed.

6.1.3 Fact Collection

- Implicit constraints between a reference and its base type (`&'a T => T: 'a`) are not collected.
- The collection of the `loan_killed_at` fact is simplified.
- Drop and unwind-related handling is not implemented due to incomplete support elsewhere in the borrow checker.
- Two-phase borrowing[18] is unaddressed. Refer to sec. 6.1.1.
- The reasons for loan invalidation are not stored, which is necessary for practical error reporting.
- Rustc prioritizes subset facts for displaying more relevant errors. This is not implemented in gccrs.

6.1.4 Polonius FFI and Error Reporting

- The current integration with the build system is not viable for production. Refer to the beginning of this chapter for details.
- Only information about the presence and category of violations is passed back to the borrow checker; details about the violations themselves are not.
- Errors are reported only at the function level (and using debug output), which can be problematic for automated testing if tests fail or succeed for incorrect reasons.

6.2 Building, Usage, and Debugging

This section provides references and basic information on how to build gccrs and use the borrow checker, along with tips for debugging.

The latest source code is available in the author's fork[19] on the branch `borrowck-stage2`[20].

[16]https://doc.rust-lang.org/nightly/reference/lifetime-elision.html#lifetime-elision
[17]https://doc.rust-lang.org/reference/trait-bounds.html#higher-ranked-trait-bounds
[18]https://rustc-dev-guide.rust-lang.org/borrow_check/two_phase_borrows.html
[19]https://github.com/jdupak/gccrs/
[20]https://github.com/jdupak/gccrs/tree/borrowck-stage2

Detailed instructions for building gccrs are in the `README.md` file in the project's root directory. For tips on a better development experience (e.g., faster builds), refer to [19].

The compiler binary is named `crab1`, and it is located in the `gcc` directory within the chosen build directory after building. Since gccrs is still experimental, the flag `-frust-incomplete-and-experimental-compiler-do-not-use` is required to use the compiler. To enable the borrow checker, add the `-frust-borrowcheck` flag. Any detected borrow checker errors will be reported as standard compilation errors.

```
$ crab1 -frust-incomplete-and-experimental-compiler-do-not-use \
      -frust-borrowcheck some_rust_code.rs

../../gcc/testsuite/rust/borrowck/borrowck-assign-comp.rs:5:1:
error: Found loan errors in function a
5 | fn a() { // { dg-error "Found loan errors in function a" }
| ^~
```

For a deeper inspection of the borrow checker, a debug build of the compiler is necessary. The `-frust-debug` flag enables debug logs, including the borrow checker activity. Unfortunately, GCC's debug logging lacks category filtering. The reader may find the variance analysis log, the borrow checker log (BIR build and fact collector), and Polonius debug output particularly interesting. This flag also activates the BIR dump (saved to `./bir_dump/<crate_name?>/<function_name?>.bir.dump`) and *facts* dump (saved to `nll_facts_gccrs/<function_name>.facts`).

```
crab1: note: Variance analysis solving started:
crab1: note: Variance analysis results:
crab1: note:  Point<>
```

```
../../gcc/testsuite/rust/borrowck/borrowck-assign-comp.rs:5:1: note:
 Checking function a

    5 | fn a() { // { dg-error "Found loan errors in function a" }
      | ^~
crab1: note: BIR::Builder::build function={a}
crab1: note:  ctx.fn_free_region={}
crab1: note:  handle_lifetime_param_constraints
crab1: note: visit_statemensts
crab1: note:  Sanitize constraints of Point{Point {x:isize, y:isize}}
crab1: note:  _4 = BorrowExpr(_1)
crab1: note:      push_subset: '?2: '?1
crab1: note:  _5 = Assignment(_6) at 0:5
crab1: note:  _9 = Assignment(_8) at 0:7
crab1: note:  _0 = Assignment(_10) at 0:11
crab1: note:  Sanitize field .0 of Point{Point {x:isize, y:isize}}
crab1: note:  Sanitize deref of & Point{Point {x:isize, y:isize}}
```

```
crab1: note:  Sanitize field .0 of Point{Point {x:isize, y:isize}}
```

To obtain similar output from rustc, use the flags `-Znll-facts -Zdump-mir=nll -Zidentify-regions`. With a debug build of rustc, you can also enable the borrow checker debug log using the environment variable `RUSTC_LOG=rustc_borrowck`. Building rustc is described in the Rustc Developer Guide[21].

For more advanced debugging and inspection, gdb/lldb can be used as usual. A common issue with LLDB is its difficulty in correctly identifying virtual classes. To address this, a simple LLDB formatter for resolving TyTy classes based on internal identifiers is available in this gist[22]. This script can be used as a template and can be adapted to other classes suffering from this problem.

[21]https://rustc-dev-guide.rust-lang.org/building/how-to-build-and-run.html
[22]https://gist.github.com/jdupak/68af0f0ad91f3e6eba2c478dc4f662dd

Chapter 7
Conclusion

This project aimed to implement a prototype of a Polonius-based borrow checker for Rustc GCC to explore the feasibility of this approach and establish a code infrastructure for further development. The development was conducted in a personal fork[1] of Rust GCC, and stabilized parts are being integrated into the main Rustc GCC GitHub repository[2]. All accepted changes are scheduled to be integrated into the central GCC repository[3] by the maintainers of Rust GCC with the help of the author.

This text described the problem of borrow checking, mapped the situation in rustc and gccrs, and presented the design of the solution, as well as the experiments that led to it. The prototype version of the implemented borrow checker can detect most common errors in simple Rust code. These include violations of access rules (number of allowed loans of a given type, loan/access conflicts), move/initialization errors, and subset errors. Examples of detected errors can be found in the appendix.

The last chapter provides an overview of the prototype's limitations. These limitations are not fundamental and should be resolvable with simple extensions and implementation of missing cases in the existing code. Future work should address these limitations to provide a production-ready solution.

Given the complex nature of borrow checking, a comprehensive, fully functional solution is likely to take months, if not years, of future work. This project provides a significant stepping stone toward a production-ready solution, offering extensive infrastructure for further development and solutions to most of the challenging problems identified in the analysis.

I believe that the Rust programming language will play a significant role in systems programming, and I would like to continue working on this project, on other problems in Rustc GCC, or on the rustc compiler itself. There appears to be considerable interest in the industry as well. Bradley Spengler, President of Open Source Security, Inc., one of the two main sponsors of Rust GCC, has expressed interest in financially supporting my continued work on Rust GCC.

[1]https://github.com/jdupak/gccrs/
[2]https://github.com/Rust-GCC/gccrs
[3]https://gcc.gnu.org/git/

Appendix A

References

[1] N. D. Matsakis and F. S. Klock, "The rust language," in *Proceedings of the 2014 ACM SIGAda annual conference on high integrity language technology*, Oct. 2014. doi: 10.1145/2663171.2663188[1].

[2] N. Matsakis, "Polonius: Either borrower or lender be, but responsibly." Rust Belt Rust Conference, Jan. 2020. Accessed: Jan. 04, 2024. [Online]. Available: https://www.youtube.com/watch?v=_agDeiWek8w

[3] R. Rakic and N. Matsakis, "Polonius update." Oct. 2023. Accessed: Jan. 04, 2024. [Online]. Available: https://blog.rust-lang.org/inside-rust/2023/10/06/polonius-update.html

[4] A. Cohen, "The road to compiling the standard library with gccrs." EuroRust, 2023. Accessed: Jan. 05, 2024. [Online]. Available: https://www.youtube.com/watch?v=WgqGahDl-sY

[5] *"Software Memory Safety" Cybersecurity Information Sheet.* The National Security Agency, 2022. Accessed: Dec. 29, 2023. [Online]. Available: https://media.defense.gov/2022/nov/10/2003112742/-1/-1/0/csi_software_memory_safety.pdf

[6] Mitre, "CWE-416: Use after free." Accessed: Dec. 20, 2023. [Online]. Available: https://cwe.mitre.org/data/definitions/416.html

[7] N. Matsakis, "2094-nll," in *The Rust RFC Book*, Rust Foundation, 2017. Accessed: Dec. 18, 2023. [Online]. Available: https://rust-lang.github.io/rfcs/2094-nll.html

[8] A. Stjerna, "Modelling Rust's Reference Ownership Analysis Declaratively in Datalog," Master's thesis, Uppsala University, 2020. Accessed: Dec. 28, 2023. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1684081/fulltext01.pdf

[9] N. Matsakis, "Polonius revisited, part 1." Sep. 22, 2023. Accessed: Dec. 17, 2023. [Online]. Available: https://smallcultfollowing.com/babysteps/blog/2023/09/22/polonius-part-1/

[10] N. Matsakis, R. Rakic, *et al.*, "The Polonius Book." Rust Foundation, 2021.

[11] *Reference.* LLVM Project, 2023. Accessed: Dec. 15, 2023. [Online]. Available: https://llvm.org/docs/Reference.html

[12] R. M. Stallman and the GCC Developer Community, *GNU Compiler Collection Internals*, 14th ed. Free Software Foundation, 2023. Accessed: Dec. 18, 2023. [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/

[13] *Rust Compiler Development Guide.* Rust Foundation, 2023. Accessed: Dec. 18, 2023. [Online]. Available: https://rustc-dev-guide.rust-lang.org/index.html

[14] Rustc developers, *Reference.* Rust Foundation, 2023. Accessed: Dec. 07, 2023. [Online]. Available: https://doc.rust-lang.org/reference/

[15] “#compiler-development > Borrowchecking vs (H)IR - GCC rust - zulip.” Sep. 05, 2023. Accessed: Dec. 05, 2023. [Online]. Available: https://gcc-rust.zulipchat.com/#narrow/stream/281658-compiler-development/topic/Borrowchecking.20vs.20.28H.29IR

[16] 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: 10.1145/1993316.1993569[2].

[17] “Polonius.” Rust Foundation. Accessed: Dec. 29, 2023. [Online]. Available: https://github.com/rust-lang/polonius/

[18] “Rust.” Rust Foundation. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/rust-lang/rust/

[19] J. Dupák, “Contribution to the Rust front-end for the GCC compiler,” research report, Czech Technical University in Prague, 2023. Accessed: Jan. 05, 2023. [Online]. Available: https://jakubdupak.com/dev/academic/dupakjak-svp-report.pdf

[20] N. Matsakis, “Polonius revisited, part 2.” Sep. 29, 2023. Accessed: Dec. 30, 2023. [Online]. Available: https://smallcultfollowing.com/babysteps/blog/2023/09/29/polonius-part-2/

[21] A. Beingessner *et al.*, *The Rustonomicon.* Rust Foundation, 2023. Accessed: Dec. 15, 2023. [Online]. Available: https://doc.rust-lang.org/nomicon/

Appendix B
Rustc Intermediate Representations Examples

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

B.1 Rust Source Code

```
struct Foo(i32);

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

B.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:3:8: 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,
        },
    ),
}
```

B.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),
}
```

B.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 C
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.

C.1 Compilation Commands

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

C.2 Rust Source Code

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

C.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);
```

C.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;
    }
}
```

```
        _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 D
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.

D.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 occurence of the error depends on runtime values. Error is raised bacause 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;
    }
}
```

D.2 Subset Errors

TODO

D.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 E
Glossary

ABI	Application Binary Interface
3-AD	Three Address Code
API	Application Programming Interface
AST	Abstract Syntax Tree
BIR	(gccrs) Borrow-Checker Intermediate Representation
CFG	Control Flow Graph
CLI	Command Line Interface
GCC	GNU Compiler Collection
GENERIC	(GCC) The internal representation used by GCC as an interface between the front-end and the middle-end of the compiler
GIMPLE	(GCC) The internal representation used by GCC in the middle-end of the compiler
HIR	(rustc, gccrs) High-level Intermediate Representation
IR	Intermediate Representation
LLVM	Low Level Virtual Machine
MIR	(rustc) Mid-level Intermediate Representation
MIRI	(rustc) The Rust MIR interpreter
NLL	(rustc) Non-Lexical Lifetimes (a CFG-based borrow-checker)
Polonius	The name of the new borrow-checker algorithm and engine
RAII	Resource Acquisition Is Initialization (C++ idiom)
RFC	Request For Comments (formal process for proposing changes to Rust)
SSA	Static Single Assignment
THIR	(rustc) Typed High-level Intermediate Representation
TyTy	(rustc, gccrs) Type Intermediate Representation (used after types are parsed and resolved)
basic block	A sequence of instructions with a single entry point and a single exit point
borrow	(Polonius) The act of taking a checked reference
fact	(Polonius) Information about the program, reduced to a relation between enumerated program objects
gccrs	GCC Rust Front-end
interning	The process of replacing a value with a unique identifier
loan	(Polonius) The result of a borrow operation (taking a checked reference).

origin	(Polonius) An inference variable that represents a set of loans. May be used interchangeably with *region*.
outlives	(Polonius) A relationship between two origins, where the first region must live longer than the second region. Denoted as `R1: R2` where `R1` outlives `R2`. That means that the set of CFG points R1 represents must be a superset of the set of CFG points R2 represents.
point	(Polonius) A point in the CFG
region	(Polonius/NLL) An inference variable that represents a set of points in the CFG. May be used interchangeably with *origin*.
rustc	The main Rust Compiler based on LLVM
usize	Unsigned integer type with the same size as a pointer in Rust