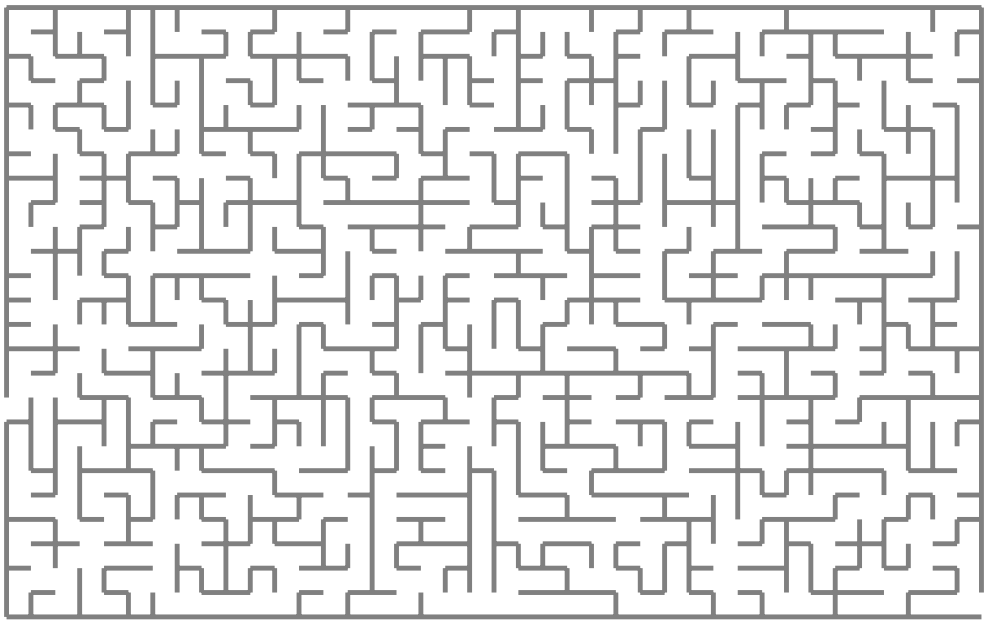
A Static analysis for Rust in Infer

Version of April 13, 2026



Arjan Seijs

A Static analysis for Rust in Infer

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Arjan Seijs
born in Amsterdam, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: Random maze.

A Static analysis for Rust in Infer

Author: Arjan Seijs
Student id: 4594045

Abstract

Rust is a systems programming language that guarantees both performance and Memory Safety through its memory model. With unsafe rust it is possible to opt-out of some security guarantees made by the compiler. However, this comes at the risk of re-introducing memory bugs.

While various tools already exist for detecting memory errors in (unsafe) Rust, they are either dynamic or focus on proving the absence of bugs. In this thesis we lay the foundations for a static analysis for Rust that focuses on proving the presence of bugs (Under Approximation) instead of their absence (Over Approximation). We show how we can use a formal framework called incorrectness separation logic (ISL) to show the presence of memory errors in Rust. We add Rust support to Infer, a static analysis tool by Meta that uses ISL to prove the presence of bugs. This is done by translating Rust to a Verification Intermediate Language. Finally, we show that our extension to *c*an be used to detect memory bugs in Rust such as *null pointer dereferences*, *dangling pointer dereferences* and *use after frees*.

Thesis Committee:

Chair:	Dr. S. Chakraborty, Faculty EEMCS, TU Delft
Committee Member:	Dr. M.A. Costea, Faculty EEMCS, TU Delft
Committee Member:	Dr. S. Dumančić, Faculty EEMCS, TU Delft

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traditional systems programming languages like C and C++ give the programmer a lot of control on how memory is managed. This, however, introduces risks of memory safety issues when not handled properly such as *use after free* bugs or *dangling pointers*. Microsoft estimates that around 70% of the security issue found are caused by memory safety issues [1].

Rust [22, 41] is a systems programming language that has gained significant popularity over the past few years, and support for it is now being added to the Linux kernel [32]. Like C and C++, the Rust language was designed to map directly to hardware, giving users control over the running time and memory usage of their programs. "This implies, for example, that all Rust types can be allocated on the stack and that Rust never requires the use of a garbage collector or other runtime [23]".

In contrast to C and C++, Rust allows low-level systems programming while making guarantees about memory safety, eliminating entire classes of bugs, such as *dereferencing invalid memory*, *data races* and *use after free*, without the need for a garbage collector. These guarantees are achieved by Rust's borrow checker that enforces a strict ownership and borrowing model at compile time. In Rust, values have only one owner. When ownership is moved the original variable is invalidated. To be able to reuse values without changing ownership, a reference can be created. A reference is similar to a pointer in that it is an address we can follow to access the data stored at that address; that data is owned by some other variable. The borrow checker keeps track of ownership and scopes of variables to make sure references point to valid memories [33].

To see how Rust prevents memory safety bugs, take the example code in Listing 1.1. When the function is called, memory is allocated on the heap for the string. This memory should be deallocated when the memory is no longer needed. In C-like languages this needs to be done manually by the programmer, which can be error-prone. The memory can be freed too early while it is still needed, causing a *use after free* bug, or forgotten to be freed, causing a *memory leak*.

Listing 1.1: Storing a string on the heap

```
1 fn heap_string() {
2   let s = String::from("hello");
3 }
```

Take as another example the code in Listing 1.2, where a pointer to a stack variable is returned. After the function call, the stack frame may be already cleaned up or overwritten by the call to another function. Dereferencing this pointer will then lead to undefined behavior. Rust keeps track of the scopes of the values and throws a compile error when it detects that a reference points to a variable that has gone out of scope.

One limitation of Rust is that the Rust Compiler is conservative. It prefers to reject some valid programs rather than risk accepting some invalid programs. The compiler rejects code

Listing 1.2: A Dangling Pointer prevent by Rust

```
fn foo() {
  let x = 10;
  let ptr = &x;
  return ptr
}
```

if it is not confident enough that the code is correct even though the code still might be valid in respects to Rust's memory guarantees. With unsafe Rust with combination with raw pointers it is possible to bypass.

Raw pointers are closer to pointers in languages like C. They are not checked by the borrow checker and can be dangling or aliased. Since a raw pointer is not guaranteed to point to valid memory, it is only possible to dereference them in parts of code that are unsafe. Additionally, unsafe Rust allows programmers to interface with external code. It is now up to the programmer to prevent these bugs by making sure that only pointers that point to valid memory are dereferenced and additionally that the memory follows the aliasing rules defined by the borrowing model. This is, however, prone to errors. Improperly managing the memory in unsafe Rust can lead to the exact memory errors Rust is trying to prevent. The program may then display *undefined behavior* such as *null pointer dereferences*, *dangling pointer dereferences* and *use after frees*.

Detecting such undefined behavior in Rust has been the subject of various academic research [34, 4, 3, 21]. For example: Miri is a dynamic analysis tool for detecting undefined behavior in Rust, and has been integrated into the continuous integration of Rust's standard library [34, 17]. Kani is a verifier that can be used to detect correctness of a program and find some cases of undefined behavior [43]. Rudra is a tool for detecting undefined behavior in unsafe Rust through the detection of common error-prone patterns [6].

However, these approaches for detecting undefined behavior do come with drawbacks. Miri can only detect undefined behavior at runtime. Kani requires the creation of a test suite to write proof guards. And although, in the case of RustBelt, proving correctness is useful, its limitation is that, similarly to that of the Rust compiler, it is conservative, e.g. it produces false positives.

Static analysis has had a great impact industry-wide in improving code quality. Its uses vary from linters that use static analysis to highlight common error patterns in code, to formal methods that prove properties about the code. One such static analysis tool that is of interest is Infer. Infer is an abstract interpretation engine for programming languages like C, C++, Java, and more. It is used within Meta as part of it its Continuous Integration (CI) pipeline to verify select properties of its projects. *h*as multiple analysis engines. One of the analyzers is Pulse. It is an interprocedural memory safety analysis to prove the presence of memory errors such as *use-after-free*, *dangling pointers* and *null pointer dereferencing* [29].

The Pulse analysis is based on incorrectness separation logic (ISL) [30] which is a formal logic system for proving the presence of memory bugs. It is a combination of incorrectness logic (IL) [25] and separation logic (SL)[37].

SL is a logical framework based on Hoare logic that facilitates reasoning about the safe use of computer memory. IL is a logic framework that is used to reason about the presence of bugs using under-approximation instead of over-approximation.

ISL combines them to prove the presence of memory bugs. In ISL, reasoning is done on small parts of a program, it only considers the local memory that is used by this component without considering the state of the whole program. Reasoning will be done only on the part of the heap a command actually access. This is called local reasoning. It then composes

those small components together using a procedure called bi-abduction [37]. SL provides the *points to assertion* $l \mapsto v$ which states that a location or pointer l contains the value v [8].

While various research has been done on applying SL to Rust, at this point there has not been done a lot of research into applying ISL to Rust. In RustBelt [18, 16] SL is used to prove soundness of the Rust type system, and various standard library functions where checked. Currently, ISL is extensively used by Meta as parts of its continuous integration to detect memory errors in languages such as C, and C++.

In this thesis we have taken the first steps towards static analysis of Rust programs using under-approximation instead of over-approximation. I.e. proving incorrectness (proving the presence of bugs) instead of correctness (proving the absence of bugs). We identified how ISL can be used to detect memory errors in (unsafe) Rust and added Rust support to Infer's Pulse analysis. To add support to Pulse, we extend $w$ith a new frontend that translates Rust files to an Intermediate Representation (IR) called Textual. The Textual IR can then be used by Pulse to detect memory errors.

In Figure 1.1 an overview is given of the stages of the static analysis with the contribution of our the frontend highlighted in green.

Figure 1.1: Translation Steps Memory Bug Detection

In this thesis we investigate how to prove the presence of classical memory bugs such as null pointer dereferences, dangling pointer dereferences and use after frees with the use of incorrectness separation logic (ISL) and work towards a static analysis tool for Rust that uses under-approximation instead of over-approximation

In this thesis the following contributions are made:

- Identify which fragment of Rust can be analyzed with existing ISL;
- Show how current ISL can be used to detect classic memory errors in Rust, such as *null pointer dereferences*, *dangling pointer dereferences* and *use after frees*;
- Show the current limitations of ISL for Rust;
- A syntactical translation from Rust to Infer's IR called Textual to detect memory errors in unsafe Rust, and implementing the translation in Infer;
- Showing the use of this translation by showing examples of errors currently detectable by our analysis;

In Chapter 2 an introduction to Rust will be given. It will be shown how Rust prevents certain types of memory errors and show how using unsafe Rust can reintroduce them. In Chapter 3 an overview of ISL is given, and it is shown how ISL can be used to detect memory bugs. Chapter 4 is the core chapter of this thesis and gives a high level overview of how Rust support is added to Infer, and shows how we translate Rust to Infer's IR Textual. Then finally in Chapter 5 we will evaluate our analysis by running it on a selection of small Rust programs.

Chapter 2

Rust

This chapter gives an introduction to the Rust programming language. In Section 2.1, ownership is explained, and it is shown how that is used for automatically cleaning up memory. Then in section Section 2.2 we show how you can reuse a variable without taking ownership. Section 2.3 talks about the mutability of references and variables. Then in Section 2.4, we show how the borrowchecker prevents certain memory errors in Rust. In Section 2.5 we show how unsafe Rust can be used to get around of some of the limitations of the borrowchecker followed by showing in Section 2.6 how this can lead to memory errors if not handled carefully. Then we close this chapter in Section 2.7.1 with an overview of the Rust compiler.

2.1 Ownership

Ownership is a unique Rust feature that handles how memory is managed. Programs need to allocate memory to store data and deallocate that data when it is no longer needed so that other allocations can use that part of memory again. In most low level languages, this allocation and deallocation needs to be done manually. This can be error-prone and lead to undefined behavior such as *null pointer dereferences*, *dangling pointer dereferences* and *use after frees*. *Use-after-free-bugs* occur when data is deallocated too early. This data can now be overwritten by other parts of the program, which can lead to bugs and security vulnerabilities.

Languages like Java, Python and C# handle memory management by the use of garbage collectors, allocating and deallocating memory for the programmer automatically. Garbage collectors run alongside a program to check whether data in memory is still used. When it detects a piece of data that is no longer used, it will deallocate the data [12].

However, running alongside the program can make affect performance and also make it hard to predict when exactly memory will be deallocated, which can result in unpredictable performance drops when the garbage collector decides to clean up the memory.

Rust takes a different approach to memory management. It does not rely on a garbage collector but instead relies on ownership to detect when memory is no longer being used. Ownership is defined by the following three rules:

- Each value in Rust has an owner.
- There can only be one owner at the time.
- When the owner goes out of scope, the value will be dropped, and the memory used by it will be freed.

With the ownership model, the ownership of a value can be transferred by a *move*. Such a move occurs when assigning a variable to another variable, when passing a value as an

argument to a function, or when returning a value from a function. Only when a variable that has ownership of a value goes out of scope, the value will be dropped and its memory freed. To show why this is needed, take for example the code in Listing 2.1. On line 2 a string is created in the current scope. This is done by allocating memory on the heap and storing a pointer to that data in the variable s. Then on line 3 the current scope ends, and Rust will automatically free the memory allocated for the string. However, this approach can give issues if not handled carefully when introducing another variable. Take for example the code in Listing 2.2. On line 2 there will be again memory allocated for the contents of the string, then on line 3 we assign the variable $s1$ to $s2$. On line 4, both variables go out of scope. Without an ownership model, it is not immediately clear which of the two variables need to be dropped. Dropping both $s1$ and $s2$ will free the memory allocated for the string twice leading to a *double free bug*. The ownership model prevents this bug. When the string is created the ownership is assigned to $s1$. When $s1$ is assigned to $s2$ on line 3, its ownership is *moved* to $s2$. Now when the scope ends on line 3, only $s2$ owns the value, and therefore only $s2$ needs to be dropped. When ownership is moved the initial variable is invalidated and cannot be used again.

Listing 2.1: Variable scope

```
1     {
2         let s = String::from("hello");
3         // s is dropped here
4     }
```

Listing 2.2: Moving ownership of the string

```
1     {
2         let s1 = String::from("hello");
3         let s2 = s1
4         // only s2 is dropped here
5     }
```

Listing 2.3: Invalidating of moved value

```
1     let s1 = String::from("hello");
2     let s2 = s1; //s1 is moved to s2
3
4     println!("{s1},␣world!"); //error: s1 has moved and is no longer valid
```

For functions, ownership works similarly, when a value is passed to a function its ownership will move inside that function, and when a value is returned from a function the ownership will move outside the function to the assigned value. This is shown in Listing 2.4 [33].

Listing 2.4: Ownership passed to function

```
1  fn main() {
2      let s1 = String::from("hello");
3      let s2 = printstring(s1)
4      // Ownership moved to s2 (via function)
5      println!("{s2}␣Bob!")
6  }
7
8  fn printstring(s: String) {
9      println!("{s}␣world!")
10     s // Return s returning ownership
11 }
```

2.2 Borrowing

The limitation with the approach shown in Listing 2.4 is that when a variable is passed to a function, it needs to be returned in order to reuse it after the function call. It is quite tedious to have to explicitly return ownership. To avoid this pattern Rust allows one to *borrow* the value by creating a reference to it. References allow functions to use values without receiving ownership of the value. References are similar to pointers in languages like C, in that it is treated as an address that we can follow to access the data stored at the address. The difference with raw pointers is that references are guaranteed to always point to valid memory, in Section 2.4 it will be shown how Rust guarantees this. In Listing 2.5 the code is adjusted to use a reference instead of taking ownership.

Additionally, it is also possible to create a reference through another reference, this is called reborrowing. This is useful when a reference is needed to a part of the value instead of the whole value. Reborrowing is shown in Listing 2.6

Listing 2.5: References

```
fn main() {
    let s1 = String::from("hello");
    printstring(&s1)
    println!("{s1}␣Bob!")
}

fn printstring(s: &String) {
    println!("{s}␣world!")
}
```

Listing 2.6: Reborrowing

```
 let x = 10;
 let r1 = &x;
 let r2 = &*r1;
```

2.3 Mutability

In the examples discussed so far, all the variables declarations are immutable. In Rust variables are immutable by default. The advantage of this is that values cannot be accidentally changed. In order to indicate that a variable is mutable the keyword *mut* can be used. Similar to variables, references are also immutable by default. However, mutable references can only borrow mutable variables, and when there is a mutable reference to a value, no other references to that value are allowed during the lifetime of the mutable reference. In Section 2.4 it is shown why this is useful. An example of mutable references is given in Listing 2.7. On line 3 a mutable reference y is created to the mutable variable x, and used to mutate its value. Since the last use of y is on line 4, it is possible to create another mutable reference later on line 6. This is only possible since the scope of the two mutable references does not overlap. In Section 2.5 an example will be given where this is the case.

Listing 2.7: Mutable Variables and references

```
let mut x = 10;
x += 1;
let y = &mut x;
*y += 1;
println!("{x}"); //Prints 12
```

```
let z = &mut x;
*z += 1;
println!("{x}"); //Prints 13
```

2.4 Borrow Checker

In Section 2.2, it was shown how references can be used to borrow a value without taking ownership. In this section, the borrow checker is introduced to show how Rust guarantees that references do not introduce memory errors. In order to prevent memory errors, it is important that references always point to valid memory. During compile time, Rust's borrow checker enforces restrictions on how references can be used.

The borrow checker enforces that references are used properly based on lifetime analysis. Every reference has a lifetime associated with it, which is the scope for which that reference is valid. Lifetimes are most of the time implicit and inferred by Rust. See for example Listing 2.8, the function *foo* returns a pointer to a stack variable. After the function call the stack is cleaned up and the returned pointer will be dangling. The borrow checker detects this and prevents the program from compiling.

In Section 2.3 it was already mentioned that there can be at most one mutable reference to some value. The benefit of restricting the amount of mutable references, is that data races and certain cases of use-after-free bugs are prevented at compile.

An example of how an *use-after-free* is prevented is shown in Listing 2.9. At line a vector with 1 elements is created, which stores a list of 3 values on the heap. Then two references are created, one mutable references to the whole vector, and one reference to the element at index 2 in the vector. When on line 6 a new vector is assigned through the mutable reference, the data of the old vector will be deallocated and v2 would point to invalid memory. [33, 39].

Listing 2.8: Example Dangling Reference Prevented by the Rust Compiler

```
fn foo() {
    let x = 10;
    let ptr = &x;
    return ptr
}

fn main() {
    let ptr = foo();
    *ptr = 20 // Invalid
}
```

Listing 2.9: Use after Free Prevented by the rust compiler

```
let mut v = vec![0, 1, 2];
let vref = &mut v;
let v2 = &v[2];

*vref = vec![4,5, 6];
let x = *v2;
```

2.5 Unsafe

One limitation of Rust is that the compiler is conservative. It prefers to reject some valid programs rather than risk accepting some invalid programs. The compiler rejects code if it

is not confident enough that the code is correct even though the code still might be valid in respects to Rust's memory guarantees.

This is where unsafe Rust comes in together with the use of raw pointers. In Rust it is possible to mark parts of the code as unsafe. In unsafe Rust it is for example possible to:

- dereference a raw pointer,
- call an unsafe function or method,
- access or modify a mutable static variable

Raw pointers are similar to references in that they point to some data on the stack or heap. However, they differ from references in a few aspects, as they [42]:

- Are allowed to ignore the borrowing rules by having both immutable and mutable pointers or multiple mutable pointers to the same location.
- Aren't guaranteed to point to valid memory.
- Are allowed to be null.

Using unsafe Rust does not necessarily mean that the code will cause undefined behavior, the responsibility is only moved from the borrow checker to the programmer. The programmer must ensure that the program does not cause memory errors. Additionally, it is still expected that, even though the borrow checker does not enforce it, the borrow model for aliasing is still followed.There does not yet exist am official formal semantics for what is considered a violation of the alias model in unsafe Rust. There are two proposed models: Stacked Borrows and Tree Borrows [19, 44, 42].

To see why unsafe Rust sometimes is needed see Listing 2.10. Here a mutable array is created, and two mutable references are created to two different parts of the array. As discussed in Section 2.4, this does not compile. Although the mutable references are to different parts of the array, therefore to different values, the borrow checker can't deduce this and rejects the code. In Listing 2.11 the same code is given using raw pointers instead of mutable references. Now it is possible to mutate the values through an unsafe block.

Listing 2.10: Multiple mutable references prevented by the compiler.

```
let mut v = [1, 2, 3];
let a = &mut v[0];
let b = &mut v[1];

*a += *b;
*b += *a;
```

Listing 2.11: Multiple mutable references using unsafe Rust.

```
let mut v = [1, 2, 3];

let a = &raw mut v[0];
let b = &raw mut v[1];

unsafe {
  *a += *b;
  *b += *a;
}
```

An additional reason for unsafe Rust is for interfacing with Foreign Function Interface (FFI) for hardware or other programming languages. If Rust didn't allow unsafe operations, it would not be possible to do low-level systems programming, such as directly interacting with the operating system [42]. Since hardware is inherently unsafe, and external code does not necessarily follow the Rust guarantees. It is up to the programmer to make sure the guarantees are held.

2.6 Undefined Behavior

In Section 2.5 it was shown how raw pointers could be used to "bypass" the borrow checker and why this is sometimes needed. In this section, it will be shown how bypassing the borrow checker in unsafe rust could introduce undefined behavior if not handled carefully.

While Rust does not allow references to be *null*, it is possible for raw pointers to be null. This can be useful when working with performance critical low level data-structures that use raw pointers. A null pointer then indicates that the value does not exist. Additionally, when interfacing with FFI for hardware or other languages, pointers passed to and from the FFI may also be null. When it is not carefully checked that a pointer is null for dereferencing it can lead to the program to crash. In Listing 2.12 an example is given of a program that panics when dereferencing a *null-pointer*.

Listing 2.12: Null pointer dereference.

```
fn main() {
    let ptr: *const i32 = std::ptr::null();
    let x = unsafe {*ptr};
}
```

In Section 2.4 it was shown how the borrow checker prevents dangling pointers. In Listing 2.13 the same code is given but using raw pointers instead of references. Since the raw pointers are not checked by the borrow checker, a dangling pointer will be created to the stack variable.

Listing 2.13: Dangling pointer dereference.

```
fn foo() -> *mut i32  {
    let mut x = 10;
    let ptr = &raw mut x;
    return ptr
}

fn main() {
    let ptr = foo();
    unsafe {*ptr = 20} // Invalid
}
```

In Listing 2.14 it is shown how in unsafe Rust a use after free bug occurs when not properly handling the ownership of variables in combination with raw pointers. In the function *pointer* a box is created, a box is a kind of pointer for data allocated on the heap. When the variable that owns the box goes out of scope it will be dropped and the memory allocated on the heap will be deallocated. In Listing 2.14 a box is created on line 2. Then on line 3, this box is then reborrowed as a raw pointer and returned. At the end of the function the variable x goes out of scope and the memory will be freed. Since a raw pointer is returned instead of an owned value or a reference, the borrow checker does not detect that the pointer outlives the value it references, therefore the pointer will be dangling.

Listing 2.14: Use after free in unsafe rust

```
fn pointer() -> *const i32 {
    let x = Box::new(50);
    &*x //Reborrow as raw pointer
    ^ drop(x)
}

fn main() {
    let result = pointer();
    let undefined = unsafe {*result};
}
```

In Listing 2.15 it is shown how a double (or triple) free bug can occur through the use of raw pointers. On line 2 a box is created, then on line 3 another box is created contain this box and its ownership is moved. On line 4 this box is casted to a raw pointer circumventing and on lines 5 and 6 the pointer is dereferenced through the use of `std::ptr::read(ptr)`, which dereferences a value from pointers without moving the ownership of the value in the pointer. This results in that b1, b2 and bbox all have ownership of the box created on line 2.

Listing 2.15: Double free in unsafe rust

```
fn main() {
    let b = Box::new(50);
    let bbox = Box::new(b);
    let bptr: *const Box<i32> = &*bbox;
    let b1 = unsafe { std::ptr::read(bptr) };
    let b2 = unsafe { std::ptr::read(bptr) };
    ^ drop(b2)
    ^ drop(b1)
    ^ drop(b)
}
```

2.7 Rust Compiler

The Borrow Checker described in Section 2.4 is an important step in the compile process of Rust programs to guarantee memory safety. It operates on an IR for Rust called Mid-Level Intermediate Representation (MIR). In this section it will be explained what MIR is, and shown how *rustc* the Rust compiler works.

Before arriving at MIR the Rust compiler performs various intermediate steps. After parsing, *rustc* will lower the Abstract Syntax tree (AST) to High-Level Intermediate Representation (HIR). HIR is a compiler-friendly representation of the AST. Here the whole program is type checked and lowered to Typed High Level Intermediate Representation (THIR), which is similar to HIR but with all type information made explicit. This will then be further lowered to MIR, and finally MIR will be lowered to the LLVM IR and passed on to LLVM to compile to machine code. An overview of this is given in Figure 2.1 [27, 20].

2.7.1 Mid-Level Intermediate Representation (MIR)

MIR is the last IR of Rust before the code is translated into LLVM IR and passed on to LLVM for machine code generation. In MIR Rust is reduced to a core set of primitives. It makes all types explicit. transforms method calls into fully quantified function calls. Additionally, it makes all drops explicit. Control-flow is simplified by modelling MIR as a Control flow

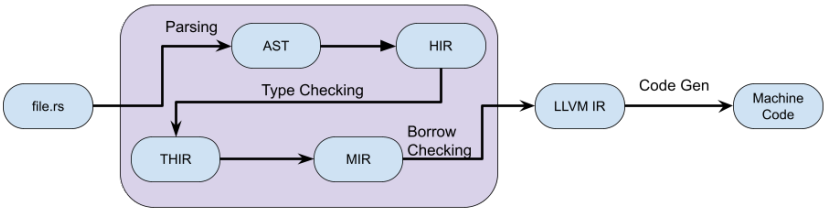


Figure 2.1: The Rust Compiler

Graph (CFG). This construction makes MIR a popular IR for analysis of Rust programs, and is used among others by Miri [34], Prusti [28], Kani [43], and Rudra [6].

Internally MIR is represented as a set of data structures that encode a CFG. It is a set of basic blocks connected by edges, where a block consists of statements. Each block ends in a terminator that perform a single action, and they define how the blocks are connected. Control-flow such as loops, if-else statements, and match expressions are represented by the block and the edges between the blocks.

MIR models the memory of a program with *places*. A place is either a local variable or a projection on another place, such as accessing a field of an object or dereferencing a memory location. Locals variables are memory locations on the stack, such as function arguments and temporaries, they are identified by a de Bruijn Index. Expressions that produce a value over places or constant values are known as *rvalues*. The *rvalues* are used in *statements* that perform a single action over them. Such actions are for example assigning a *rvalue* to a *place* or dropping a *rvalue* [40, 27].

Chapter 3

Incorrectness Separation Logic

ISL is a formal framework that facilitates reasoning about mutations to computer memory for proving the presence of bugs. It is a combination of SL and IL. In this chapter we will show how ISL can be used to detect memory errors in Rust. First in Section 3.1 and Section 3.2 we will show SL and IL respectively.

3.1 Separation logic

Separation logic (SL) is a mathematical logic framework for reasoning about mutations to computer memory. It enables scalability by using local reasoning about the memory of small chunks of programs and then composing the reasoning chunks together.

SL uses a logical connective $*$ called the *separating conjunction* for logic formulae that are interpreted over program allocated heaps. The logical formula $(A * B)$ represents a piece of a program heap, when it can be divided into two independent heaps described by A and B. In this way the logic framework mirrors that of mutation to computer memory, reasoning about program commands work by updating $(*)$-conjucts in place. For example, the formula $x \mapsto y * y \mapsto x$ can be read as two allocated memory cells x and y, where x points to y and y points to x. Since the heaps are independent, i.e. there is no variable in multiple conjuctions pointing to different values, updating one of the memory cells won't have influence of the other.

SL uses Hoare triples of the form $\{pre\}program\{post\}$, where *pre* is the precondition and *post* is the post condition describing the abstract behavior of a program. Since heaps are independent we can use the separating conjunction in combination to go from smaller specifications to bigger specifications. This is called the Frame Rule (Equation (3.1)) of SL and is the key principle of local reasoning in SL. The frame in Equation (3.1) describes the portion of memory that remains unchanged by executing the program.

$$\frac{\{pre\}program\{post\}}{\vdash \{pre * frame\}program\{post * frame\}}\ \text{frame-rule} \tag{3.1}$$

Take for example the specification for a resource that can be open or closed in Equation (3.2). If there are two resources r_1 and r_2, and we close the first of them, we can use the Frame Rule to get from a small specification to a bigger specification [37, 31, 25].

$$[r \mapsto open]closeResource(r)[r \mapsto closed] \tag{3.2}$$

$$[r_1 \mapsto open * r_2 \mapsto open]closeResource(r_1)[r_1 \mapsto close * r_2 \mapsto open] \tag{3.3}$$

3.1.1 Abduction

Abductive Inference in standard logic is the process of finding a missing assumption that makes an entailment true. For example, given an assumption A and a goal G, abduction tries to find a missing assumption M that makes the entailment $A \wedge M \vdash G$ true.

Similarly, abductive inference can be used for SL to generate preconditions. Take for example a procedure that has a precondition $G \stackrel{\text{def}}{=} A * B$, and at the site where G is called the assertion A holds. This assertion A alone does not match up with the precondition of G, additional conditions need to be found such that it matches the given precondition. "Formally this involves solving the question of the form $A * ?? \vdash G$" [8].

Algorithm 1 Abduce

procedure Abduce(Δ, H)
  Find M such that $\Delta * M \vdash H$
  return (M)
end procedure

3.1.2 Bi-abduction

Bi-abduction is an extended form of abduction and is the backbone for reasoning in SL. In addition to synthesizing the missing portion of state, bi-abductionalso synthesizes the left-over portions of state. This way it automates the logical inference by slowly building the required pre- and post-conditions for a program. Bi-abduction tries to answer the question which *frame* and *antiframe* make the following question valid:

$$A * antiframe \vdash B * frame$$

The question can be answered by appealing to separate frame inference and abduction procedures. Bi-abduction first calls regular abduction as described in section 3.1.1 to find a missing antiframe.

Then it calls a given procedure called *Frame*. This procedure finds a "Leftover" heap L. The procedure must satisfy:

$$Frame(H_0, H_1) = L \Rightarrow H_0 \vdash H_1 * L$$

The Bi-abduction is given in Algorithm 2 with a Bi-abduction version of the Frame Rule given in Equation (3.4) [8, 37].

$$\frac{[A]C[B]BiaAbd(P, A) = (M, L)}{\vdash [P * M]C[B * L]} \text{ frame-rule-bi-abduction} \tag{3.4}$$

Algorithm 2 Bi-abduction

procedure BiAbd(Δ, H)
  $antiframe \leftarrow Abduce(\Delta, H * true)$
  $frame \leftarrow Frame(\Delta * antiframe, H)$
  return $(antiframe, frame)$
end procedure

3.2 Incorrectness logic

In IL under-approximation is used to prove the presence of bugs, in contrast to Hoare Logic which uses over-approximation to prove the absence of bugs. Similarly to Hoare Logic we use under approximation triples in the form of $[presumption]code[result]$. The result is a subset of all the states that can be reached satisfying the presumption in contrast to Hoare logic where the post condition would be a superset of the states. "Every state in the result is reachable from some state in the presumption" [25].

In IL reasoning can be done by focusing on fewer paths as seen in Equation (3.5). Dropping the disjunction allows of strengthening the post condition, which allows to reason about each path independently and get two possible specifications for a function [25].

$$\frac{[p]C[q_1 \vee q_2]}{\vdash [p]C[q_1]} \text{ drop-disjunction} \tag{3.5}$$

3.3 Incorrectness separation logic

ISL combines SL and IL. It uses bi-abduction to proof the presence of bugs in programs. Similarly to IL, ISL uses under approximation triples in the form of $[presumption]code[result]$. The original SL is incompatible with local, under-approximate reasoning, and needs to be extended by adding a negative heap assertions: $x \not\mapsto$which states that x has been deallocated.

The use of bi-abduction in ISL is even better suited than in SL. In SL adding a missing heap using the Frame Rule work only for straight-line code and not across control flow branches, as there is no guarantee that a safe precondition for one path is safe for the other. In contrast, in ISL it will check both paths independently and will generate pre- and post-conditions for both those paths. If one of those paths produces an error it will generate a post condition with a latent error that is only manifested when its preconditions are met.

Take for example the following c code:

Listing 3.1: Latent error in pulse

```
1  void null_if_positive(int n, int** p)
2  {
3
4      if (n > 0)
5      {
6          *p = NULL;
7      }
8  }
9
10 void null_dereference(int n, int** p) {
11   null_if_positive(n, p);
12   **p = 42;
13 }
```

In the function `null_if_positive` there is a conditional branch on $n > 0$. In ISL both branches will be traversed independently and two pre post conditions will be generated for the function:

1. $[n > 0 \wedge n \mapsto N * p \mapsto -][Ok : n \leqslant 0 \wedge n \mapsto N * p \mapsto NULL]$
2. $[n \leqslant 0 \wedge n \mapsto N * p \mapsto -][Ok : n \leqslant 0 \wedge n \mapsto N * p \mapsto -]$

Then in `null_dereference` the `null_if_positive` procedure is called, and a value is stored in the pointer p. The above generated specifications are then used to now generate pre- and

post-conditions for null_dereference, one of which results in an error because of a null pointer dereference:

1. $[n > 0 \land n \mapsto N * p \mapsto -][Err : n \leqslant 0 \land n \mapsto N * p \mapsto NULL]$
2. $[n \leqslant 0 \land n \mapsto N * p \mapsto v_1 * v_1 \mapsto -][Ok : n \leqslant 0 \land n \mapsto N * p \mapsto v_1 * v_1 \mapsto 42]$

The error in the first post-condition is a latent error since the error will only manifest when the procedure is called with the matching pre-condition. The set of ISL triples in pulse we used is shown in Figure 3.1 [37, 30].

3.4 ISL Proof Rules

This sections shows ISL can be used to detect the memory errors discussed in Section 2.6. They are of the form $[presumption]code[result]$ as described in Section 3.3.

Listing 3.2 exemplifies how a null pointer dereference can be detected using ISL. The function *std::ptr::null* from the Rust standard library is annotated with a summary describing its functionality (line 1). Calling this function on line 5 results in the abstract program state described on line 6 which reflects the binding of `ptr` to the result returned by the function *null*. The dereference of `ptr` in line 7 leads to an *null-pointer-derefence* detected with ISL by applying the Load-Null rule, as reflected in the abstract program state at line 8.

Listing 3.2: Null pointer dereference detected with ISL

```
1 [emp]std :: ptr :: null[Ok : return = null]
2
3 fn main() {
4     [emp]
5     let ptr: *const i32 = std::ptr::null();            (Assign)
6     [Ok : ptr = null]
7     let x = unsafe {*ptr};                             (Load Null)
8     [Err : ptr = null]
9 }
```

In Listing 3.3 it is shown how a *use-after-free* can be detected. On line 1 a summary of the pre- and post-conditions of the Rust library function `Box::new` are given. One line 4 the start condition is the empty heap. Calling the `Box::new` on line 5 results in the return value being assigned to x, showing on line 6 the abstract state were the variable x points to the value Fifty. The pre-condition will be updated through the Frame Rule. On line 7 the value will be reborrowed as a raw pointer and is set to be returned at the end of the function. The notation `&*` indicates that the value first is dereferenced and then a reference of that value is created. However, in reality Rust treats this as a direct copy and casts this value from a box type to a pointer type, this is why it is treated as a `Assign` instead of a combination `Store` and `Load`. On line 9 the box goes out of scope and its value will be automatically dropped, invalidating the memory. The function `pointer` will then have the following pre- and post-conditions.

$$[emp]pointer()[Ok : return \not\mapsto]$$

Listing 3.4 shows how ISL can detect double frees. On line 1 the pre- and post-conditions of the `std::ptr::read` function int the Rust library is given. This function dereferences a pointer without moving the ownership of the value. This makes the `Read` rule similar to that of the `Load` rule. On line 12 and 14 this function is used to get two owned values to the box created at line 5. They will therefore both be dropped when their scope ends. On line 17 $b2$ is dropped invalidating the pointer and consequently with the `Cons` rule $b1$ will also be invalidated leading to the error condition on line 20 by the `Free Err` rule.

Listing 3.3: Use after free detected with ISL

```
1  [emp]Box :: new(e)[Ok : return ↦ e] (BoxNew)
2
3  fn pointer() -> *const i32 {
4      [emp]
5      let x = Box::new(50);                    (Assign) (BoxNew)
6      [Ok : x ↦ 50]
7      &*x                                      (Assign)
8      [Ok : x ↦ 50 * return = x]
9      ^ drop(x)                                (Free)
10     [Ok : x ↛ * return = x]                  (Cons)
11     [Ok : return ↛ * return = x]
12 }
13
14 fn main() {
15     [emp]
16     let result = pointer(); (Assign) (Pointer)
17     [Ok : result ↛ ]
18     let undefined = unsafe {*result}; (Load Err)
19     [Err : result ↛ ]
20 }
```

Listing 3.4: Double free detected with ISL

```
1  [ptr ↦ e]std :: ptr :: read(ptr)[Ok : ptr ↦ e * return = e] (Read)
2
3  fn main() {
4      [emp]
5      let b = Box::new(50);                                  (Assign) (BoxNew)
6      [Ok : b ↦ 50]
7      let bbox = Box::new(b);                                (Assign) (BoxNew)
8      [Ok : b ↦ 50 * bbox ↦ b]
9      let bptr: *const Box<i32> = &*bbox;                    (Assign)
10     [Ok : b ↦ 50 * bbox ↦ b * bptr = bbox]
11     [Ok : b ↦ 50 * bptr ↦ b * bptr = bbox]                 (Cons)
12     let b1 = unsafe { std::ptr::read(bptr) };              (Assign) (Read)
13     [Ok : b ↦ 50 * bptr ↦ b * bptr = bbox * b1 = b]
14     let b2 = unsafe { std::ptr::read(bptr) };              (Assign) (Read)
15     [Ok : b ↦ 50 * bptr ↦ b * bptr = bbox * b1 = b * b2 = b]
16     [Ok : b2 ↦ 50 * bptr ↦ b * bptr = bbox * b1 = b * b2 = b] (Cons)
17     ^ drop(b2)                                             (Free)
18     [Ok : b2 ↛ * bptr ↦ b * bptr = bbox * b1 = b * b2 = b]
19     [Ok : b1 ↛ * bptr ↦ b * bptr = bbox * b1 = b * b2 = b] (Cons)
20     ^ drop(b1)                                             (Free Err)
21     [Err : b1 ↛]
22     ^ drop(b)                                              (Seq)
23     [Err : b1 ↛]
24 }
```

$$[Alloc][x = x']x = alloc()[x \mapsto -]$$

$$\vdash [x \mapsto e][x] := y[Ok : x \mapsto y] \text{ Store}$$

$$\vdash [x = null][x] := y[Err : x = null] \text{ Store Null}$$

$$\vdash [x \not\mapsto][x] := y[Err : x \not\mapsto] \text{ Store Er}$$

$$\vdash [x = x' * y \mapsto e]x := [y][Ok : x = e * y \mapsto e] \text{ Load}$$

$$\vdash [y = null]x := [y][Err : y = null] \text{ Load Null}$$

$$\vdash [y \not\mapsto]x := [y][Err : y \not\mapsto] \text{ Load Er}$$

$$\vdash [x = x']x := e[Ok : x = e[x'/x]] \text{ Assign}$$

$$\vdash [x \mapsto e]free(x)[Ok : x \not\mapsto] \text{ Free}$$

$$\vdash [x \mapsto e]free(x)[Err : x \not\mapsto] \text{ Free Error}$$

$$\vdash [x = null]free(x)[Err : x = null] \text{ Free Null}$$

$$\frac{p' \Rightarrow p \ \vdash [p']C[\epsilon : q'] \ q \Rightarrow q'}{\vdash [p]C[\epsilon : q]} \text{ Cons}$$

$$\frac{\vdash [p]C_1[Ok : r] \ [p]C_2[\epsilon : q]}{\vdash [p]C_1; C2[\epsilon : q]} \text{ Seq Ok}$$

$$\frac{\vdash [p]C_1[Err : q]}{\vdash [p]C_1; C2[Err : q]} \text{ Seq Err}$$

$$\frac{\vdash [p]C[\epsilon : q]}{\vdash [p * r]C[\epsilon : q * r]} \text{ Frame}$$

Figure 3.1: Pulse ISL Triples [30]

Chapter 4

Implementation

This chapter discuses the implementation details together with some of the design decisions that were made. We start by offering a high overview of the extensions made to Infer. Then in Section 4.1 we will give an overview of an IR similar to MIR, which we will be using for translating Rust. In Section 4.2 a will give a brief overview of the grammar of Textual. Finally in Section 4.3 we will show how we translate the IR to Textual.

To add Rust support for Infer we add a new frontend that translates Rust to Textual. Textual is a clone of smallfoot intermediate language (SIL). It is designed to be able to be written easily by hand, and requires less redundant information. It is used for writing small unit tests and models, and is designed to make adding new frontends to Infer easier. Textual is modeled as a CFG [26].

Section 2.7 showed which stages the Rust compiler goes through when translating Rust to machine code and gives an overview of the IRs in the various stages of the translation. MIR is the last IR before code generation is passed on the LLVM. It is a popular IR for writing analyzers. At first sight MIR is also promising for translating Rust to Textual. Both MIR and Textual are CFGs which makes mapping the constructs in both languages easier.

However, there are some drawbacks when using MIR directly for the translation. Developing a new tool targeting Rust currently requires a significant work effort to interact with the Rust compiler. The various IRs in rustc are optimized for speed and efficiency, not ease of consumption by analysis tools. Instead of providing a complete data-structure that can be used it provides queries to obtain only part of expression when needed. Program information is stored in various representations, sometimes too low-level to be directly usable. For example, constants in MIR may already be compiled and stored as an array of bytes. Structs definitions are only stored at HIR level and at MIR level handled the same way as tuples losing the field name information.

The Charon [14] library address this problem by creating an analysis framework for Rust. It takes care of the more tedious parts of interfacing with the Rust compiler providing a data structures that can serve as the foundation of analyzes. "Charon constructs a cleaned-up, decorated view called Unstructured Low-Level Borrow Calculus (ULLBC)" [14]. ULLBC is similar to MIR in that they are both CFGs. The difference is that ULLBC offers contextual and semantic information without needing to query to the Rust compiler. For example, in ULLBC constants are simplified, and user defined structs and fieldnames are reconstructed [14].

We add a new frontend Infer called Rust Driver. The frontend takes ULLBC as input and translate this to Textual. Which will then be used by Infer to generate a bug report. In Figure 4.1 we give an overview of all the steps we go through when analyzing a rust file with our contribution highlighted in Green. In Section 4.1 gives an overview of ULLBC and shows some differences between ULLBC and MIR will be shown. Section 4.2 will give an overview of the Textual syntax. Then in Section 4.3 we show how ULLBC is translated to Textual by going through some example translations.

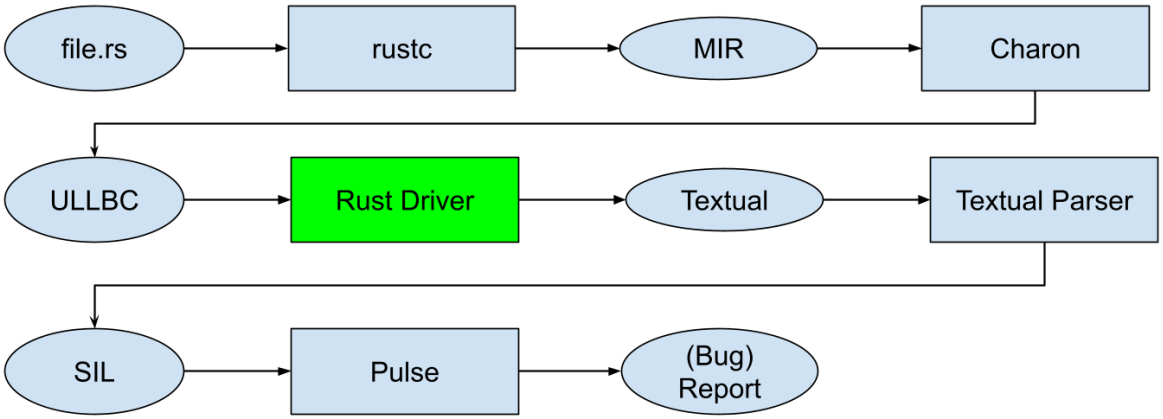


Figure 4.1: Translation Steps

4.1 ULLBC

ULLBC is a cleaned up version of MIR that offers a more structured semantic view of MIR. Similar to MIR, ULLBC is represented as a set of data-structures that encode a CFG with additional information about users types that have been reconstructed.

Listing 4.1 shows an example program of how Charon reconstructs type information. This program creates a struct that represent a point with an x and y value which are then assigned to variables. Listing 4.2 show that in MIR the type decoration given at the point of instantiation, and when accessing its field, the type name information is lost. Additionally, variable name information has also been lost. Listing 4.3 shows the ULLBC of the same program. Here it is shown that the user defined type is included and that the field names are reconstructed.

Listing 4.1: Example Rust program

```
fn main() {
    let point = Point {x : 10, y : 20};
    let x = point.x;
    let y = point.y;
}

struct Point {
    x: i32,
    y: i32
}
```

Listing 4.2: MIR

```
fn main() -> () {
    let mut _0: ();
    let _1: Point;
    let _2: i32;
    let _3: i32;
    bb0: {
        _1 = Point { x: const 10_i32, y: const 20_i32 };
        _2 = copy (_1.0: i32);
        _3 = copy (_1.1: i32);
        return;
    }
```

```
}
```

Listing 4.3: ULLBC

```
struct Point {
  x: i32,
  y: i32,
}

fn main()
{
    let @0: (); // return
    let point@1: Point; // local
    let x@2: i32; // local
    let y@3: i32; // local
    let @4: (); // anonymous local

    bb0: {
        point@1 := Point { x: const (10 : i32), y: const (20 : i32) };
        x@2 := copy ((point@1).x);
        y@3 := copy ((point@1).y);
        @0 := ();
        return;
    }
}
```

The code in Listing 4.4 shows how Charon reconstruct types for functions with generic parameters. The function *id* takes and returns ownership of a value of a generic type. This is then called with an integer and a character. In Listing 4.5 the MIR of the program is given. In MIR the type information is given at the function call. However, only one specific instance of the *id* function is generated. Listing 4.6 shows how Charon generates type specific implementations of the *id* function where all type information is made explicit.

Listing 4.4: Rust

```
fn id<T>(x : T) -> T {
    x
}

fn main() {
    let one = id(1);
    let a = id('a');
}
```

Listing 4.5: MIR

```
fn id(_1: T) -> T {
    bb0: {
        _0 = move _1;
        return;
    }
}

fn main() -> () {
    bb0: {
```

```
        _1 = id::<i32>(const 1_i32) -> [return: bb1, unwind continue];
    }
    bb1: {
        _2 = id::<char>(const 'a') -> [return: bb2, unwind continue];
    }
    bb2: {
        return;
    }
}
```

Listing 4.6: ULLBC

```
fn main()
{
  ...
    bb0: {
        one@1 := id::<i32>(const (1 : i32)) -> bb1
    }
    bb1: {
        a@2 := id::<char>(const (a)) -> bb3
    }
    bb3: {
        @0 := ();
        return;
    }
}

fn id::<i32>(@1: i32) -> i32
{
    let @0: i32; // return
    let x@1: i32; // arg #1

    bb0: {
        @0 := move (x@1);
        return;
    }
}

fn id::<char>(@1: char) -> char
{
    let @0: char; // return
    let x@1: char; // arg #1

    bb0: {
        @0 := move (x@1);
        return;
    }
}
```

Figure 4.13a describes (a subset of) the ULLBC grammar. Each function in ULLBC is a set of basic blocks connected by edges, where a block consists of statements. Each block ends in a terminator that performs a single action, and they define how the blocks are connected. Variables in ULLBC are defined by a place. A place is either a de Bruijn Index, or a projection

on another place, such as taking a reference, deference, index or field. The two statements that are used in the translation are *Drop* and *Assign*. The drop statement frees up memory from heap stored at the location stored in the place. The assign statement $p := rv$ assigns a *rvalue* to a place. *Rvalues* are expressions that produce a value.

4.2 Textual

Textual is the IR used by Infer. Textual is one layer of abstraction above the SIL, the IR on which Infer's analysis is executed. The goal of Textual is to make it easier for adding support of new languages to Infer. Textual and SIL are represented as a CFG. This can be seen in Textual Figure 4.13b. Textual and SIL differentiate between logical variables, and program variables. Program variables represent the address on the stack or heap of a variable of the original program. You cannot take an address of a logical variable, and a program variable can only be manipulated through its address [26].

A Textual program is a collection of function procedures. Each procedure contains multiple nodes, and each node contains instructions followed by a terminator. There are two instructions used in the translation. The let expression $id? = e$ evaluates an expression and optionally assigns this to a logical variable. The store instruction **store** $e_1 \leftarrow e_2$ takes two expressions and stores the value of the expression e_2 evaluates to at the address where expression e_1 evaluates to.

4.3 Translation Rules

This section shows how ULLBC is translated to Textual. The translation rules are given in Figure 4.16 through Figure 4.24 with the signatures of the rules given in Figure 4.14. The translation uses two types contexts. Γ_{label} is a context that maps block labels in ULLBC to node labels in Textual. Γ_{place} is a context that maps local variables represented by their de Bruijn Indices in ULLBC towards a program variable in Textual.

4.3.1 Places

In this section we show how a place is translated from ULLBC to Textual. The full set of rules for translating places are shown in Figure 4.16. A place in ULLBC is either a variable represented by a de Bruijn Index, or a projection on another place such as accessing a field of an object, or dereferencing a pointer. Additionally, ULLBC reconstructs the original variable names from the Rust program and also includes the type information of the place.

We translate a place to a Textual expression and its Rust type to equivalent Textual type.

$$\Gamma_{place} : p \xrightarrow[\text{place}]{} (e,\ type)$$

Figure 4.2 gives an example translation from a variable in Rust to a Textual `LVar` (`Local Place`). The `LVar` in Textual correspond to a variable in Rust. However, SIL, and by extension Textual, uses an indirection in how it handles variables. Instead of directly modifying the values of variables, they are only handled through its address. The `LVar` is therefore the address of a program variable. This is indicated by the `&` . In Section 4.3.2 and Section 4.3.3 this will be shown in more detail how this indirection is handled when actually reading and assigning values to and from variables.

Rust

```
let x
```

ULLBC

```
x@1
```

Textual

```
&x_1
```

Figure 4.2: Rust Variable to Program Variable

The translation of projections on places is a bit more straight forward, since Textual has similar constructs for dereferencing, indexing arrays, and accessing fields. We show this in Figure 4.3 were the translation of dereferencing a variable x shown `(Deref Place)`. The equivalent expression is called a `Load` in Textual and is indicated by square brackets.

Rust

```
*x
```

ULLBC

```
*x@1
```

Textual

```
[&x_1]
```

Figure 4.3: Derefrence to Load expession

4.3.2 Operands

This section describes how ULLBC operands are translated to Textual expressions. The rules for this translation can be found in Figure 4.17. An operand is either a constant value, or a move or copy of some value stored at a place. We translate an operand to a Textual Expression and a Textualtype.

$$\Gamma_{place} : operand \xrightarrow[\text{op}]{} (e,\ type)$$

The translation of the constant operand to a constant in Textual (`Constant`) is quite straightforward, since ULLBC already handled simplifying the constants. This is shown in Figure 4.4.

Rust

```
10
```

ULLBC

```
const (10 : i32)
```

Textual

```
10:int
```

Figure 4.4: Rust Constant to Textual Const

The translation of the copy and move operands require a bit more care (`Copy, Move`). In Rust, the move and copy operands are operations on places that make a bitwise copy of the value stored at the place. However, we cannot directly assign the Textual expression retrieved from translating the place, due to the indirection of how variables are treated as Textual discussed in Section 4.3.1. The translated expression is the address where the value of the Rust program variable is stored. This value can be retrieved by inserting an extra load instruction. Since the Rust compiler already enforces the proper use of moves and copy, it is not needed to encode this information again at the Textual level. Figure 4.5 shows an example program of how an assign instruction with the copy operand is translated to textual. The assign instruction will be explained in more detail in Section 4.3.3

Rust

```
1  let y = x;
2
```

ULLBC

```
1  y@2 := copy (x@1);
2
```

Textual

```
1  store &y_2 <- [&x_1:int]:int
2
```

Figure 4.5: Copy to Textual

4.3.3 Statements

This section shows how ULLBC statements are translated to Textual instructions. These translation rules are shown in Figure 4.18 There are two statements in ULLBC that are relevant: The assign statement and the drop statement.

$$\Gamma_{place} : stmt \xrightarrow[\text{stmt}]{} instr$$

Figure 4.6 shows a Rust program is shown that sets the variable y to the value of variable x. Since the types of the variables are primitives it is shown in the ULLBC that it copies the value instead of a move. The assign statement is then translated to a store instruction (`Assign`). In Textual the left-hand side of a store instruction takes an expression, the expression on the left-hand side resolves to a value that will be the address where the value of the right-hand expression is stored. On the first line in Textual, the value 10 is stored at the address referenced by the program variable &x_1. Then, on the second line, the value stored at the address is loaded and assigned to be stored at the address of program variable &y_2.

Rust

```
1  let x = 10;
2  let y = x;
3
```

ULLBC

```
1  x@1 := 10;
2  y@2 := copy (x@1);
3
```

Textual

```
1  store &x_1 <- 10: int
2  store &y_2 <- [&x_1:int]:int
3
```

Figure 4.6: Assign to Store

Figure 4.7 shows an example a program that creates a box type. This is an owned pointer to some value on the heap. How box types are precisely translated will be shown later in Section 4.3.4. Here we will show how ULLBC makes the drop of this explicit. On line 3 of the MIR code the implicit drop of box is made explicit. In the Textual we show on line 2 how this is translated to an equivalent inbuilt function call in Textual (`Drop`).

Rust

```
1     let x = Box::new(10);
2     ...
3
4
```

ULLBC

```
1   x@1 := @BoxNew<i32>(const (10 : i32))
2   ...
3   drop @x1
4
```

Textual

```
1    ...
2   _ = __sil_free([&x_1:*int])
3
```

Figure 4.7: Drop Statement

4.3.4 Rvalues

Figure 4.19 shows the translation of a rvalue into a textual expression. The rvalues are the right-hand side of an assign instruction and are expressions that produce a value over some operand. We already implicitly used the *"use"* rvalue in Section 4.3.3, which directly lifts an operand to a rvalue. We translate ULLBC rvalues to Textual expressions and types.

$$\Gamma_{place} : rvalue \xrightarrow[\text{rvalue}]{} (e,\ type)$$

To see how Unary and Binary operators are translated, see the example in Figure 4.8. A Unary or Binary operator is an operation over one or two operands respectively. These are, for example, operations such as addition, subtraction, negations, and logical operations. Since in Textual Unary and Binary operators are modeled as function calls, to translate it is needed to find the function name corresponding to the operator and then translate it to a function call expression. In Figure 4.8 a binary operator is translated to a function call (`BinOp`). First we translate the binary operation to the corresponding function name in Textual (`Add Int`). Then we translate the operands, which are in this case constants to Textual constants (`Const`).

Rust

```
1     let x = 1 + 2;
2
```

ULLBC

```
1   x@1 := const (1 : i32) panic.+ const (2 : i32);
2
```

Textual

```
1     store &x_1 <- __sil_plusa_int(1, 2):int
2
```

Figure 4.8: BinOp Rvalue

References and raw pointers are translated are both translated to Textual pointers. Due to the indirection of program variables, references can be translated without needing the load expression since a program variable already is the address of the value. This can be seen on Line 2 of the Textual code in Figure 4.9. Taking the reference of a variable is her translated to storing the Lvar `&x_1` at `&_ptr` (`Ref Local`).

Rust

```
1     let x = 10;
2     let ptr = &x;
3
```

ULLBC

```
1   x@1 := const (10 : i32);
2   ptr@2 := &x@1;
3
```

Textual

```
1     store &x_1 <- 10:int
2     store &ptr_2 <- &x_1:*int
3
```

Figure 4.9: Reference Rvalue

4.3.5 Terminators

Figure 4.21 shows how terminators in ULLBC are translated to terminators in Textual. In both cases a Terminator is the last instruction of a block with either a successor or a return out of the function. However, there are differences in what kind of instruction is handled as its own terminator or a separate statement followed by another terminator.

$$\Gamma_{label}\Gamma_{place} : terminator \xrightarrow[\text{term}]{} \Gamma'_{label} : (instr+,\ term)$$

.

It takes as input a context of places and labels, and the ULLBC Terminator. This is then mapped to a list of Textual instructions and a Textual Terminator.

Figure 4.10 shows how the return terminator is translated to Textual. The assign statement on line 6 and terminator on line 7 in ULLBC shows that a function always implicitly

returns the variable with a de Bruijn Index of zero. In Textual it is possible to return complex expressions directly. To translate this call the variable at place zero needs to be translated to an expression and this will then be returned. Due to the indirection of variables the expression is surrounded by a load expression (`Return`). In ULLBC a function always return a value. Functions without a return type specified return a Unit value.

Rust

```
fn twenty() -> i32 {
  20
}

```

ULLBC

```
fn twenty() -> i32
{
    let @0: i32; // return

    bb0: {
        @0 := const (20 : i32);
        return;
    }
}

```

Textual

```
define twenty() : int {

  local var_0: int

  #node_0:
      store &var_0 <- 20:int
      ret [&var_0:int]

}

```

Figure 4.10: Return Terminator

In ULLBC a function call is a terminator that contains a lot of information and does multiple things simultaneously. First, it consists of a name of the function that is being called. It also contains a place where the result of the function will be stored. It contains the operands that are passed as arguments to the function, and finally it contains an identifier *idx* of a successor block. To translate it to Textual it needs to be translated to a separate instruction and terminator (`Call`). This is shown in Figure 4.11. The place, operands, and function name are translated to a store instruction that assigns the result of the function call to the expression corresponding with the place. The identifier *idx* is translated to a node label used in Textual's jump terminator.

Rust

```rust
fn main() {
    let x = twenty();
    let y = x + 1;
}

```

ULLBC

```
fn main()
{
     bb0: {
         x@1 := twenty() -> bb1
     }

     bb1: {
         @0 := ();
         return;
     }
}

```

Textual

```
define main() : void {
  #node_0:
       store &x_1 <- twenty()
       jmp node_1

  #node_1:
       store &var_0 <- null:void
       store &var_0 <- null:void
       ret [&var_0:void]
}

```

Figure 4.11: Call Terminator

In Section 4.3.3 we already showed how a drop statement was translated and the translation of box was introduced. We will now be shown how this box is translated in more detail. The creation of boxes is done by calling the `Box::new` function. The translation of this statement is therefore not different from that of other function calls (`Call`). Instead, we will now show how we modeled box types and the `Box::new` function. Currently, the translation supports boxes for primitives type that use a default allocator. In Rust types can have custom allocator's defined for them that handle how memory is allocated on the heap. For primitives types this is simply reserving some memory on the heap similar to a malloc in the C language and then storing the value at that location in memory. The challenge is now in how to translate this to Textual. To support boxes for primitive types a custom function needs to be defined that models the allocation of memory on the heap, stores the value on the heap ,and returns the pointer to this data. In the translation the Textual model of the *BoxNew* function is shown. The function takes a value and then returns a pointer to that value stored on the heap. On line 6 a call is made to the malloc function built into Textual. This allocates memory on the heap, and stores the address in pointer. Then on line 7, the value will be stored on the address stored in the pointer. Finally, on line 8 the pointer will be returned from the function.

Rust

```
    let x = Box::new(10);

```

ULLBC

```
  x@1 := @BoxNew<i32>(const (10 : i32))

```

Textual

```
  store &x_1 <- __sil_boxnew(10):*int

  define __sil_boxnew(value : int): *int {
  local ptr : *int
  #entry:
    store &ptr <- __sil_malloc(<int>)
    store [&ptr] <- value
    ret [&ptr]
  }

```

Figure 4.12: Box types

4.3.6 Function Declaration

In this section we show the translation of complete function declarations by revisiting an example from Chapter 3. We translate a function declaration in ULLBC to a program description in Textual (`Function`). An ULLBC function consists of:

1. Locals: The user defined variables together with variables for intermediate computations introduced by MIR. They are places defined by their de Bruijn Index.
2. Blocks: A list of statements followed by a terminator with an identifier.

First we translate all the places and ULLBC types of the local declarations to program variables and update the context Γ_{place} (`LocalDecl`). This updated context is then used in translating the blocks to nodes (`Blocks`). The statements are translated to instructions as described in Section 4.3.3 and the ULLBC-terminator to Textual terminators and a statement as described Section 4.3.5.

Listing 4.7: Rust

```
fn pointer() -> *const i32 {
    let x = Box::new(50);
    &*x
}
fn main() {
    let ptr = pointer();
    let ub = unsafe {*ptr}; // Error Occurs Here
}

```

Listing 4.8: ULLBC

```
fn pointer() -> *const i32
{
```

```
    let @0: *const i32; // return
    let x@1: Box<i32>; // local
    let @2: &'_ (i32); // anonymous local
    bb0: {
        storage_live(x@1);
        x@1 := @BoxNew<i32>(const (50 : i32)) -> bb1
    }
    bb1: {
        storage_live(@2);
        @2 := &*(x@1);
        @0 := &raw const *(@2);
        drop x@1;
        return;
    }
    bb2: {
        unwind_continue;
    }
}

fn main()
{
    let @0: (); // return
    let ptr@1: *const i32; // local
    let ub@2: i32; // local
    bb0: {
        ptr@1 := pointer() -> bb1
    }
    bb1: {
        ub@2 := copy (*(ptr@1));
        @0 := ();
        return;
    }
}

```

Listing 4.9: Textual

```
define pointer() : *int {
  local var_0: *int, x_1: *int, var_2: *int
  #node_0:
      store &x_1 <- __sil_boxnew(50):*int
      jmp node_1
  #node_1:
      store &var_2 <- [&x_1:*int]:*int
      store &var_0 <- [&var_2:*int]:*int
      _ = __sil_free([&x_1:*int])
      ret [&var_0:*int]
}

define main() : void {
  local var_0: void, ptr_1: *int, ub_2: int
  #node_0:
      store &ptr_1 <- pointer():*int
```

```
17        jmp node_1
18
19    #node_1:
20        store &ub_2 <- [[&ptr_1:*int]:int]:int
21        store &var_0 <- null:void
22        store &var_0 <- null:void
23        ret [&var_0:void]
24  }
25
```

4.3.7 ISL

In Section 3.4 it was shown how ISL could be used to detect memory errors in Rust. In this section we show how the ISL-triples can be applied to the Textual translation. In Listing 4.10 a desugared version of the Textual code in Listing 4.9 is shown. This desugaring lifts load expressions into it's separate instruction and assigns this value to a logical variable making the indirection discussed in Section 4.3.1 explicit. This makes sure that every textual instruction closely relates to a ISL rule. In Listing 4.11, Listing 4.12 and Listing 4.13 it is shown how the ISL-triples can be used to prove the *use-after-free* bug. For the sake of readability we leave usage of the `CONS` and `FRAME` rules implicit, and leave the indirection out of the state of the conjections ($[\&var \mapsto var]$).

Listing 4.10: Desugared Textual

```
1
2  define __sil_boxnew(value : int): *int {
3    local ptr : *int
4    #entry:
5      n1 = __sil_malloc(<int>)
6      store &ptr <- n1
7      n2 = load &ptr
8      store n2 <- o
9      ret n1
10   }
11
12 define pointer() : *int {
13   local var_0: *int, x_1: *int, var_2: *int
14   #node_0:
15       n0 = __sil_boxnew(50)
16       store &x_1 <- n0:*int
17       jmp node_1
18
19   #node_1:
20       n1:*int = load &x_1
21       store &var_2 <- n1:*int
22       n2:*int = load &var_2
23       store &var_0 <- n2:*int
24       n3:*int = load &x_1
25       n4 = __sil_free(n3)
26       n5:*int = load &var_0
27       ret n5
28
29 }
```

```
define main() : void {
  local var_0: void, ptr_1: *int, ub_2: int
  #node_0:
      n0 = pointer()
      store &ptr_1 <- n0:*int
      jmp node_1

  #node_1:
      n1:*int = load &ptr_1
      n2:int = load n1
      store &ub_2 <- n2:int
      store &var_0 <- null:void
      n3:void = load &var_0
      ret n3

}
```

Listing 4.11: ISL states for `boxnew` in Textual

```
[emp]
define __sil_boxnew(value : int): *int {
  [emp]
  local ptr : *int
  [&ptr ↦ ptr]
  #entry:
    n1 = __sil_malloc(<int>)        (Alloc)
    [n1 ↦ −]
    store &ptr <- n1                (Store)
    [&ptr ↦ n1]
    n2 = load &ptr                  (Load)
    [n2 = n1]
    store n2 <- value
    [n2 ↦ value]
    ret n2
    [return ↦ value]
  }
```

Listing 4.12: ISL states in Textual

```

define pointer() : *int {
  local var_0: *int, x_1: *int, var_2: *int
  [&x_1 ↦ x_1]
  #node_0:
      [emp]
      n0 = __sil_boxnew(50)
      [n0 ↦ 50]
      store &x_1 <- n0:*int
      [n0 ↦ 50 ∗ n0 = x1]
      jmp node_1

  #node_1:
      n1:*int = load &x_1
```

```
32        [n0 ↦ 50 * n0 = x1 * n1 = x1]
33        store &var_2 <- n1:*int
34        [n0 ↦ 50 * n0 = x1 * n1 = x1 * var_2 = n1]
35        n2:*int = load &var_2
36        [n0 ↦ 50 * n0 = x1 * n1 = x1 * var_2 = n1 * n2 = var_2]
37        store &var_0 <- n2:*int
38        [n0 ↦ 50 * n0 = x1 * n1 = x1 * var_2 = n1 * n2 = var_2 * var_0 = n2]
39        n3:*int = load &x_1
40        [n0 ↦ 50 * n0 = x1 * n1 = x1 * ... * n3 = x1]
41        [n3 ↦ 50 * n0 = x1 * n1 = x1 * ... * n3 = x1]
42        n4 = __sil_free(n3)
43        [n3 ↛ * n0 = x1 * n1 = x1 * ... * n3 = x1]
44        n5:*int = load &var_0
45        [n3 ↛ * n0 = x1 * n1 = x1 * ... * n5 = var_0]
46        [n5 ↛ * n0 = x1 * n1 = x1 * ... * n3 = x1]
47        ret n5
48        [return ↛]
49
50  }
```

Listing 4.13: ISL states in Textual

```
51  define main() : void {
52    local var_0: void, ptr_1: *int, ub_2: int
53    #node_0:
54        n0 = pointer()
55        [n0 ↛]
56        store &ptr_1 <- n0:*int
57        [n0 ↛ * ptr = n0]
58        jmp node_1
59
60    #node_1:
61        n1:*int = load &ptr_1
62        [n0 ↛ * ptr = n0 * n1 = ptr]
63        [n1 ↛ * ptr = n0 * n1 = ptr]
64        n2:int = load n1
65        [Err : n1 ↛]
66        store &ub_2 <- n2:int
67        store &var_0 <- null:void
68        n3:void = load &var_0
69        ret n3
70
71  }
```

$$
\begin{aligned}
\mathit{crate} &::= ((\mathit{id} : \tau)+)+\ \mathit{body}+ \\
\mathit{body} &::= \mathit{id}\ \mathit{localdecl}+\ \mathit{block}+ \\
\mathit{localdecl} &::= \mathit{mut}?\ \tau\ \_n \\
\mathit{block} &::= idx\ \mathit{stmt}*\ \mathit{terminator} \\
\mathit{terminator} &::= \mathit{goto}\ idx \\
&\mid \mathit{if}\ o\ idx_1\ idx_2 \\
&\mid \mathit{return} \\
&\mid \mathit{call}\ id\ o*\ p\ idx \\
&\mid \mathit{drop}\ p\ idx \\
\mathit{stmt} &::= \mid p := \mathit{rv} \\
&\mid \mathit{drop}\ p \\
\mathit{places} \ni p &::= \_n\ \tau \mid *p\ \tau \mid p.id\ \tau \mid p[o]\ \tau \\
\mathit{rvalue} \ni rv &::= \mathit{op}\ o*\ \tau \mid \&p \mid \&\mathsf{mut}\ p \mid *\mathsf{const}\ p \mid *\mathsf{mut}\ p \\
&\mid o* \\
\mathit{operand} \ni o &::= \mathit{copy}\ p \mid \mathit{move}\ p \mid \mathit{const}\ l\ \tau \\
\mathit{locals} &\ni \_n \\
\mathit{integers} &\ni n \\
\mathit{literals} &\ni l \\
\mathit{labels} &\ni idx \\
\mathit{operators} \ni \mathit{op} &::= Add \mid Div \mid Neg \mid Eq \mid ... \\
\tau &::= \mathit{int} \mid \mathit{float} \mid \mathit{uint} \mid \mathit{bool} \mid \mathit{char} \\
&\mid () \\
&\mid ((\mathit{id} : \tau)+) \\
&\mid (\tau+) \\
&\mid [\tau; n] \\
&\mid *\mathit{const}\ \mathit{type} \\
&\mid *\mathit{mut}\ \mathit{type} \\
&\mid \&\mathit{type} \\
&\mid \&\mathit{mut}\ \mathit{type} \\
&\mid \mathit{box}\tau
\end{aligned}
$$

(a) ULLBC syntax

$$
\begin{aligned}
\mathit{module} &::= ((\mathit{id} : \mathit{type})+)+ \\
&\quad \mathit{procedure}+ \\
\mathit{procedure} &::= \mathit{id}\ \mathit{locals}+\ \mathit{node}+ \\
\mathit{node} &::= \mathit{id}\ \mathit{instr}+\ \mathit{term} \\
\mathit{locals} &::= \mathit{id}\ \mathit{type} \\
\mathit{instr} &::= \mathbf{store}\ e \leftarrow e : \mathit{type}? \\
&\mid \mathit{id}? = e \\
\mathit{terminator} &::= \mathit{ret}\ e \\
&\mid \mathit{unreachable} \\
&\mid \mathit{jump}\ (\mathit{id}, e+)+ \\
&\mid \mathit{if}\ e \\
&\quad \mathit{then\ terminator} \\
&\quad \mathit{else\ terminator} \\
e &::= \&\mathit{lvar} \\
&\mid \mathit{const}\ n \\
&\mid e(.id)+ \\
&\mid id(e+) \\
&\mid [e : \mathit{type}?] \\
&\mid e[e] \\
\mathit{integers} &\ni n \\
\mathit{id} &\ni \mathit{lvar} \\
\mathit{type} &::= \mathit{int} \\
&\mid \mathit{float} \\
&\mid \mathit{null} \\
&\mid \mathit{void} \\
&\mid *\mathit{type} \\
&\mid (\mathit{id} : \mathit{type})+ \\
&\mid \mathit{type}[]
\end{aligned}
$$

(b) Textual syntax

Figure 4.13: Syntax of origin and target language

$$\Gamma_{\mathit{place}} : p \xrightarrow[\text{place}]{} (e,\ \mathit{type})$$

$$\Gamma_{\mathit{place}} : \mathit{operand} \xrightarrow[\text{op}]{} (e,\ \mathit{type})$$

$$\Gamma_{\mathit{place}} : \mathit{rvalue} \xrightarrow[\text{rvalue}]{} (e,\ \mathit{type})$$

$$\Gamma_{\mathit{place}} : \mathit{stmt} \xrightarrow[\text{stmt}]{} \mathit{instr}$$

$$\Gamma_{\mathit{label}}\Gamma_{\mathit{place}} : \mathit{terminator} \xrightarrow[\text{term}]{} \Gamma'_{\mathit{label}} : (\mathit{instr}+,\ \mathit{term})$$

$$\tau \xrightarrow[\text{type}]{} \mathit{type}$$

$$\mathit{op}\ \tau \xrightarrow[\text{call}]{} (\mathit{id},\ \mathit{type})$$

$$\Gamma_{\mathit{label}}\Gamma_{\mathit{place}} :\ \mathit{block} \xrightarrow[\text{block}]{} \Gamma'_{\mathit{label}} : \mathit{node}$$

$$\Gamma_{\mathit{label}}\Gamma_{\mathit{place}} :\ \mathit{body} \xrightarrow[\text{body}]{} \mathit{procedure}$$

$$\Gamma_{\mathit{label}}\Gamma_{\mathit{place}} :\ \mathit{localdecl} \xrightarrow[\text{local}]{} \Gamma'_{\mathit{place}} :\ \mathit{local}$$

$$\Gamma_{\mathit{label}} : idx \xrightarrow[\text{label}]{} \Gamma'_{\mathit{label}} : \mathit{id}$$

$$\Gamma_{\mathit{place}}(\_n) = (\mathit{id})$$

$$\Gamma_{\mathit{label}}(idx) = \mathit{id}$$

Figure 4.14: Translation Signatures

Get Label

$$\frac{\Gamma_{label}(idx) = id}{\Gamma_{label} : idx \xrightarrow[\text{label}]{} \Gamma_{label} : id}$$

Fresh Label

$$\frac{fresh\ id \qquad \Gamma'_{label} := [idx \to id]\Gamma_{label}}{\Gamma_{label} : idx \xrightarrow[\text{label}]{} \Gamma'_{label} : id}$$

Figure 4.15: Contexts

Local Place

$$\frac{\Gamma_{place}(\_n) = (id) \qquad \tau \xrightarrow[\text{type}]{} type}{\Gamma_{place} : \_n\ \tau \xrightarrow[\text{place}]{} (\&id,\ type)}$$

Deref Place

$$\frac{\Gamma_{place} : p \xrightarrow[\text{place}]{} (e,\ \_) \qquad \tau \xrightarrow[\text{type}]{} type}{\Gamma_{place} : *p\ \tau \xrightarrow[\text{place}]{} ([e],\ type)}$$

Projection Place

$$\frac{\Gamma_{place} : p \xrightarrow[\text{place}]{} (e,\ \_) \qquad \tau \xrightarrow[\text{type}]{} type}{\Gamma_{place} : p.id\ \tau \xrightarrow[\text{place}]{} (e.id,\ type)}$$

Index Place

$$\frac{\begin{array}{c}\Gamma_{place} : p \xrightarrow[\text{place}]{} (e_1,\ \_) \\ \Gamma_{place} : op \xrightarrow[\text{op}]{} (e_2,\ \_) \qquad \tau \xrightarrow[\text{type}]{} type\end{array}}{\Gamma_{place} : p[op]\ \tau \xrightarrow[\text{place}]{} (e_1[e_2],\ type)}$$

Figure 4.16: Translation Rules Places

Const

$$\frac{\tau \xrightarrow[\text{type}]{} type}{\Gamma_{place} : const\ n\ \tau \xrightarrow[\text{op}]{} (const\ n,\ type)}$$

Move

$$\frac{\Gamma_{place} : p \xrightarrow[\text{place}]{} (e,\ type)}{\Gamma_{place} : move\ p \xrightarrow[\text{op}]{} ([e],\ type)}$$

Copy

$$\frac{\Gamma_{place} : p \xrightarrow[\text{place}]{} (e,\ type)}{\Gamma_{place} : copy\ p \xrightarrow[\text{op}]{} ([e],\ type)}$$

Figure 4.17: Operands

Assign

$$\frac{\begin{array}{c}\Gamma_{place} : p \xrightarrow[\text{place}]{} (e_1,\ \_) \\ \Gamma_{place} : rvalue \xrightarrow[\text{rvalue}]{} (e_2,\ type)\end{array}}{\Gamma_{place} : p := rvalue \xrightarrow[\text{stmt}]{} \textbf{store}\ e_1 \leftarrow e_2 : type}$$

Drop

$$\frac{\Gamma_{place} : p \xrightarrow[\text{place}]{} (e,\ \_)}{\Gamma_{place} : drop\ p \xrightarrow[\text{stmt}]{} free([e])}$$

Figure 4.18: Statements

UnOp

$$\frac{op\ \tau \xrightarrow[\text{call}]{} (op_{call},\ type) \qquad \Gamma_{place} : o \xrightarrow[\text{op}]{} (e,\ \_)}{\Gamma_{place} : op\ o\ \tau \xrightarrow[\text{rvalue}]{} (op_{call}(e),\ type)}$$

BinOp

$$\frac{op\ \tau \xrightarrow[\text{call}]{} (op_{call},\ type) \qquad \Gamma_{place} : o_1 \xrightarrow[\text{op}]{} (e_1,\ \_) \qquad \Gamma_{place} : o_2 \xrightarrow[\text{op}]{} (e_2,\ \_)}{\Gamma_{place} : op\ o_1\ o_2\ \tau \xrightarrow[\text{rvalue}]{} (op_{call}(e_1, e_2),\ type)}$$

Use

$$\frac{\Gamma_{place} : o \xrightarrow[\text{op}]{} (e,\ \_) \qquad \tau \xrightarrow[\text{type}]{} type}{\Gamma_{place} : use\ o\ \tau \xrightarrow[\text{rvalue}]{} (e,\ type)}$$

Ref Local

$$\frac{\Gamma_{place}(\_n) = (id) \qquad \tau \xrightarrow[\text{type}]{} type}{\Gamma_{place} : \&\_n\ \tau \xrightarrow[\text{rvalue}]{} (\&id,\ type)}$$

Ref

$$\frac{\Gamma_{place} : p \xrightarrow[\text{place}]{} (e,\ \_) \qquad \tau \xrightarrow[\text{type}]{} type}{\Gamma_{place} : \&p\ \tau \xrightarrow[\text{rvalue}]{} (e,\ type)}$$

Raw Local

$$\frac{\Gamma_{place}(\_n) = (id) \qquad \tau \xrightarrow[\text{type}]{} type}{\Gamma_{place} : *\mathsf{const}\ \_n\ \tau \xrightarrow[\text{rvalue}]{} (\&id,\ type)}$$

Raw

$$\frac{\Gamma_{place} : p \xrightarrow[\text{place}]{} (e,\ \_) \qquad \tau \xrightarrow[\text{type}]{} type}{\Gamma_{place} : *\mathsf{const}\ p\ \tau \xrightarrow[\text{rvalue}]{} (e,\ type)}$$

Figure 4.19: Translation RValues

Add Int

$$Add\ int \xrightarrow[\text{call}]{} (\_\_sil\_plusa\_int,\ Int)$$

Equality

$$Eq\ \_ \xrightarrow[\text{call}]{} (\_\_sil\_eq,\ Int)$$

Div Float

$$Div\ float \xrightarrow[\text{call}]{} (\_\_sil\_divf,\ Float)$$

...

Figure 4.20: Translation operators to Function

Goto

$$\frac{\Gamma_{label} : idx \xrightarrow[\text{label}]{} \Gamma'_{label} : lbl}{\Gamma_{label}\Gamma_{place} : goto\ idx \xrightarrow[\text{term}]{} \Gamma'_{label} : ([],\ jump\ lbl)}$$

Return

$$\frac{\Gamma_{place} : 0 \xrightarrow[\text{place}]{} (e,\ \_)}{\Gamma_{label}\Gamma_{place} : return \xrightarrow[\text{term}]{} \Gamma_{label} : ([],\ return\ [e])}$$

If Else

$$\frac{\Gamma_{label} : idx_1 \xrightarrow[\text{label}]{} \Gamma'_{label} : lbl_1 \qquad \Gamma'_{label} : idx_2 \xrightarrow[\text{label}]{} \Gamma''_{label} : lbl_2 \qquad \Gamma_{place} : o \xrightarrow[\text{op}]{} (e,\ \_)}{\Gamma_{label}\Gamma_{place} : \textit{if } o\ idx_1\ idx_2 \xrightarrow[\text{term}]{} \Gamma''_{label} : ([],\ \textit{if } e \textit{ then jmp } lbl_1 \textit{ else jmp } lbl_2)}$$

Call

$$\frac{\Gamma_{label} : idx \xrightarrow[\text{label}]{} \Gamma'_{label} : lbl \qquad \Gamma_{place} : p \xrightarrow[\text{place}]{} (e,\ \_) \qquad \Gamma_{place} : o* \xrightarrow[\text{op}]{} (e*,\ \_)}{\Gamma_{label}\Gamma_{place} : call\ (const\ id_{fn})\ o*\ p\ idx \xrightarrow[\text{term}]{} \Gamma'_{label} : ([\textbf{store}\ e \leftarrow id_{fn}(e*)],\ jump\ lbl)}$$

Figure 4.21: Terminators

Block

$$\frac{\Gamma_{place} : stmt* \xrightarrow[\text{stmt}]{} instrs_1 \qquad \Gamma_{label}\Gamma_{place} : terminator \xrightarrow[\text{term}]{} \Gamma'_{label} : (instrs_2,\ term) \qquad instrs := instrs_1 :: instrs_2 \qquad \Gamma'_{label} : idx \xrightarrow[\text{label}]{} \Gamma''_{label} : id}{\Gamma_{label}\Gamma_{place} :\ idx\ stmt*\ terminator \xrightarrow[\text{block}]{} \Gamma''_{label} : id\ instrs\ term}$$

Figure 4.22: Basic Blocks

Function

$$\frac{\Gamma_{label}\Gamma_{place} :\ localdecl+ \xrightarrow[\text{local}]{} \Gamma'_{place} :\ locals+ \qquad \Gamma_{label}\Gamma'_{place} :\ block+ \xrightarrow[\text{block}]{} \Gamma''_{place} : node+}{\Gamma_{label}\Gamma_{place} :\ id\ localdecl+\ block+ \xrightarrow[\text{body}]{} id\ locals+\ node+}$$

LocalDecl

$$\frac{\tau \xrightarrow[\text{type}]{} type \qquad \begin{array}{c} fresh\ id \\ \Gamma'_{place} := [p \to (id)]\Gamma_{place} \end{array}}{\Gamma_{label}\Gamma_{place} :\ p\ \tau \xrightarrow[\text{local}]{} \Gamma'_{place} :\ id\ type}$$

Figure 4.23: Function

Primitive

$$prim \xrightarrow[\text{type}]{} prim$$

Unit

$$() \xrightarrow[\text{type}]{} void$$

Raw Pointer

$$\frac{\tau \xrightarrow[\text{type}]{} type}{*const\ \tau \xrightarrow[\text{type}]{} *type}$$

Mutable Raw Pointer

$$\frac{\tau \xrightarrow[\text{type}]{} type}{*mut\ \tau \xrightarrow[\text{type}]{} *type}$$

Reference

$$\frac{\tau \xrightarrow[\text{type}]{} type}{\&\tau \xrightarrow[\text{type}]{} *type}$$

Mutable Reference

$$\frac{\tau \xrightarrow[\text{type}]{} type}{\&mut\ \tau \xrightarrow[\text{type}]{} *type}$$

Array

$$\frac{\tau \xrightarrow[\text{type}]{} type}{[\tau; n] \xrightarrow[\text{type}]{} type[]}$$

Tuple

$$\frac{\tau_0 \xrightarrow[\text{type}]{} type_0 \qquad \tau_1 \xrightarrow[\text{type}]{} type_1 \qquad id_0 := 0 \qquad id_1 := 1}{(\tau_1, \tau_2) \xrightarrow[\text{type}]{} (id_0 : type_0, id_1 : type_1)}$$

Struct

$$\frac{\tau_0 \xrightarrow[\text{type}]{} type_0 \qquad \tau_1 \xrightarrow[\text{type}]{} type_1}{(id_0 : \tau_1, id_1 : \tau_2) \xrightarrow[\text{type}]{} (id_0 : type_0, id_1 : type_1)}$$

Box

$$\frac{\tau \xrightarrow[\text{type}]{} type \qquad \tau_1 \xrightarrow[\text{type}]{} type_1}{box\ \tau \xrightarrow[\text{type}]{} *type}$$

Figure 4.24: Types

Chapter 5

Evaluation

To empirically validate the static-analysis we will run the analyzer to try to answer the following questions:

- **RQ1** How common is the subset of Rust we can translate relative to the basic functionality of Rust?
- **RQ2** What class of memory-errors can the ISL for Rust detect?
- **RQ3** To what extent does the ISL-based static analysis agree with what Miri considers undefined behavior, and how does it compare against Miri with regard to finding memory errors?

We will evaluate it on three datasets: The Rust by example [15] dataset will be used to give an indication of how many of the Rust features we currently can translate. A synthetic dataset was created consisting of 57 sample programs to show how memory errors we are able to detect. Finally, we run it on a subset of the Miri test suit to see how it compares to the current baseline tools, and use that to empirically show that our analysis does not produce false positives.

5.1 Setup

For the evaluation we used the version of Infer at commit 405f202 [10]. The Charon version used is 0.1.174 [9] which uses rustc 1.95.0-nightly (efc9e1b50 2026-02-06 [35]). In this version of Infer we call Charon with the following default flags:
`--mir=elaborated`, `--precise-drops`, `--treat-boxes-as-builtin`, `--reconstruct-fallible-operations`

The flags `--mir=elaborated` and `--precise-drops` instruct Charon to retrieve the MIR after drop elaboration. Without those flags, we would get an earlier stage of MIR where drop's may be conditional. `--treat-boxes-as-builtin` gives us detailed type information for boxes. `--reconstruct-fallible-operations` tells Charon to model arithmetic without overflow checks, since this is also how arithmetic operations would be represented when Rust is translated to machine code [1].

These flags are sufficient for a lot of cases. However, sometimes the translation requires additional flags, for example:

When handling code that has generic functions, and we need precise type information, the `--monomorphize` flag is passed to Charon. This generates type specific instances for functions similar to how the Rust compiler would translate such functions into LLVM. This flag is not included by default, since the current version of Charon is not able to monomorphize

[1] In release mode

everything yet and may panic when trying to translate an unsupported function. The flags `--hide-allocator` is included when translating files that are depended on boxes . Finally, we pass specific `--include=function_name` flags when needed to also translate files from external libraries (e.g. the standard library), since by default the function body is not translated.

5.2 RQ1: Supported Subset of Rust

To get an indication of the Rust features we currently can support we evaluate our translation on a dataset extracted from Rust by Example [15]. This is a collection of runnable examples that illustrate various Rust concepts and standard libraries. In total, it contains 332 codeblocks of Rust program that cover a wide range of rust features an average Rust programmer would encounter.

5.2.1 Dataset

The program where extracted from the markdown files and included as individual files in a cargo project. At the root of the cargo project we included a custom `println!` (Listing 5.1) macro that returns a value of Unit type. This does slightly change the code since the macro does not expand the passed values to references. Without replacing this macro, we would be unable to translate all the `println` statements since they do rely on various unsupported features such as `dyn` , and a compiler intrinsic called `format_args!` that is not yet supported. From the total 332 programs 222 could be included without causing compilation errors. We run the translation on the 222 supported programs using the following combination of Charon flags in addition to the default flags described in Section 5.1:

- **Default:** No additional arguments
- **Opaque:** `--extract-opaque-bodies`
- **Monomorphized:** `--monomorphize`
- **Monomorphized & Opaque:** `--extract-opaque-bodies --monomorphize`

Listing 5.1: println macro

```
const UNIT_TYP  : () = ();

macro_rules! println {
    () => {};
    ($fmt:literal $(, $arg:expr)*) => {
        crate::UNIT_TYP
    };
}
```

5.2.2 Evaluation of Results

The results of running our translation are shown in Figure 5.1. By default, we can translate 105 of the programs without producing an error. When translating monomorphized versions this increases to 137 programs. However, using the monomorphized version of the programs also means that Charon panics on translating 37 programs to ULLBC and timeouts on 1 program. When we use `--extract-opaque-bodies` , we can translate 69 programs by default and 127 programs when monomorphized. Using this flag also extracts functions from dependencies (e.g. the standard library). Having more functions to translate by extension leads to a higher chance of encountering features that are not yet supported.

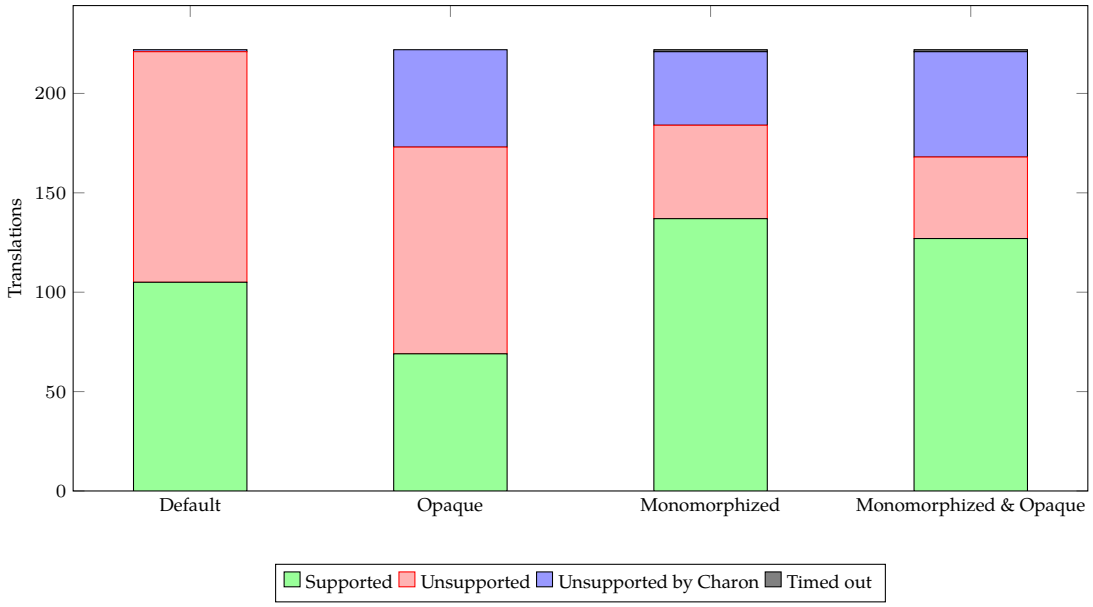


Figure 5.1: Amount of supported programs from Rust by example supported by our translation.

For the programs we cannot translate, the majority of them is because of unsupported drop types. While we do support drops for boxes of primitives types, and support some drops for compound types such as structs, other use cases of drops are not yet supported. The reason for this is that what a drop does, depends on the type that is dropped. It may implement custom drop logic or is reliant on compiler intrinsics that are not yet supported. Furthermore, when encountering drop in a function where the type is generic we do not have enough information to know how exactly this needs to be translated.

The Limitations with drops illustrate the more general case for limitations with translating generic functions. Since there is no explicit type information available yet, it is not always possible directly translate values depending on the generic directly into an expression. Take for example the code in Listing 5.2, the function `sum` is depended on the size of the array, and the actual value of `N` is unknown. Similarly, when a generic is depended on a trait method, it is unknown for which type the method needs to be called.

An overview of some of the supported features and limitations of the translation is given in Table 5.2.

What We Support	**Limitations / Future Work**
References	dyn
Raw Pointers	Non-monomorphized generics
Primitives	Compiler intrinsincs
Structs	Slices
Tuples	Advanced Drops
Arrays	
Box type for primitives	
Functions	
Drops (Partially)	

Table 5.2: Supported features for translation.

Listing 5.2: Generic Functions

```
fn sum<const N: usize>(arr: [i32; N]) -> i32 {
    let mut total = 0;
    let mut i = 0;
    while i < N {
        total += arr[i];
        i += 1;
    }
    total
}
```

5.2.3 Interpretation of Results

As we have shown above, the amount of programs we can translate depends on whether functions are monomorphized and also whether the functions from dependencies are extracted. By default, we can translate 46% of the programs, when monomorphizing we can translate 62% of the total programs or 74% of the programs that are supported by Charon. **Answer to RQ1:** *Although in the case of unmomorphized Rust programs the translation is still limited, when translating monomorphized code we can translate the majority of programs.*

5.3 RQ2: Class Of Memory Errors

5.3.1 Dataset

We create a test suite of in total 57 Rust programs to evaluate the analysizer. We run both Miri on the tests to check whether the tests we created are accurate and run our analyzer on them to see whether it gives the correct output. I.e, it detects some of the errors that Miri detects, and does not give any false-positives on the tests that Miri does not detect. The dataset contains of 4 different categories of tests:

1. 25 programs that do not display undefined behavior.
2. 22 programs were we can detect undefined behavior.
3. 7 programs that would display undefined behavior when turning of the borrow checker.
4. 3 programs of errors currently not detected by our analyzer.

5.3.2 Evaluation of Results

The test suits show that we can detect undefined behavior in the form of classical memory errors. The errors the analysis currently can detect are *null-pointer-dereferences*, *dangling-pointer-dereferences*, *use-after-lifetime*, *use-after-free* and *double-frees*. We will now go over some of undefined behavior in Rust we currently cannot detect. Rust does however not have a formal model of its semantics yet, so what is and isn't considered undefined behavior may change in the future [7].

Dangling Reference

In Listing 5.3 we show the creation of a dangling reference similar to the creation of a dangling pointer discussed in previous chapters. However, this time the pointer is cast to a reference. Our analysis can currently detect that when dereferencing the reference, this would be invalid since it is dangling. In Rust, however, the behavior is already considered undefined

when casting the raw pointer to a reference, since this breaks the Rust contract that reference always should always point to valid memory.

Listing 5.3: Dangling Reference

```
fn dangle() -> *mut i32 {
    let mut x = 10;
    let ptr = &raw mut x;
    ptr
}

fn main() {
    let ptr : &i32 = unsafe {std::mem::transmute(dangle())}; // UB occurs here
    let ub = *ptr; // UB is detected here
}
```

Variable Scopes

In Listing 5.4 a dangling pointer is created to a variable inside a scope. At the end of the scope the lifetime of the variable ends and the value will be dropped. In the case of reference Rust would detect that the references outlives the value and therefore would not compile. This is however not the case in when using raw pointers.

Listing 5.4: Variable Scope

```
fn main() {
    let ptr = {
        let x = 0;
        &x as *const i32
        //lifetime of x ends.
    };
    let _ub = unsafe{*ptr};
}
```

Stack Borrow Violation

The following code shows a violation of the stacked borrow semantics. While the program seems to behave to fine when executed, running Miri on the code produces an error when executing the program.

Listing 5.5: Variable Scope

```
fn main() {
    let ptr = freed();
    let ub = unsafe {*ptr};
    let b : Box<i32> = unsafe{std::mem::transmute(ptr)};
}

fn freed() -> *const i32 {
    let x = Box::new(50);
    let result = (&*x) as *const i32;
    std::mem::forget(x);
    result
}
```

5.3.3 Interpretation of Results

The results show that ISL is useful for detecting classical memory errors in Rust, however to detect Rust specific undefined behavior the current ISL model would need to be extended do detect violations of Rust's memory model. **Answer to RQ2:** *While the ISL analysis for Rust currently cannot detect undefined behavior arising from violations of its memory model, it does identify classical memory errors such as null pointer dereferences, dangling pointer dereferences and use after frees*

5.4 RQ3: How does the analysis compare to miri

5.4.1 Dataset

We ran the analyzer on a subset of the Miri test suite (2c90efd32). For this we have chosen the following two test folders to run our analyzer on: `/fail/dangling` contains tests for the detection of classical memory errors. This gives us an indication on the memory errors we currently can detect and how our analysis compares to Miri in finding memory errors. Furthermore, we ran the tests on the programs in `/pass/` to see whether our analysis does not provide false-positives. We run the tests suit two times, first without any additional Charon flags, and secondly with `--monomorphize` to run the analysis on monomorphized of the code.

5.4.2 Evaluation of Results

Figure 5.3 and Figure 5.4 show the results of running our analysis on the Miri test bench. We can detect 7 issues in `/fail/dangling`. Two memory issues were detected in the `/pass/` tests suite, one of them is true-positives of a memory leak issue where the Miri test suit has disabled the check for these issues (`pass/memleak_ignored.rs`). The other is a warning of a dereferencing a dangling pointer for a zero-sized object (`pass/zst.rs`). Although, it is not a false postive in the sense that this state is in reality not reached, it is in Rust not considered undefined behavior to write and read from a dangling pointer of zero-sized types.

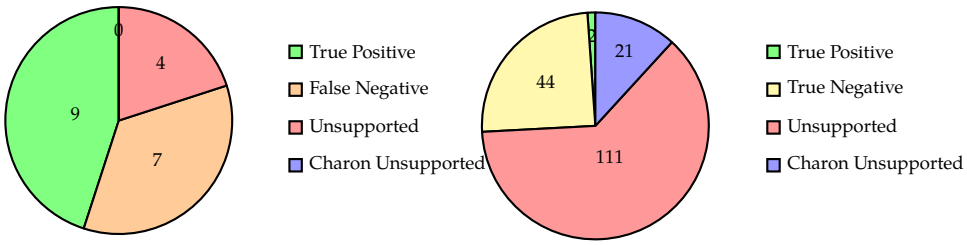


Figure 5.3: Detected errors and supported features of the Miri test suits `/fail/dangling` (Left) and `/pass/` (right)

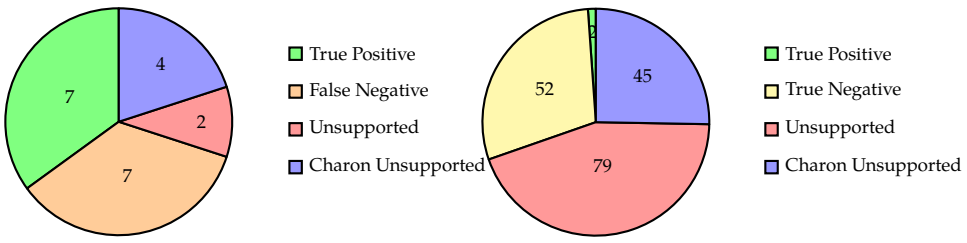


Figure 5.4: Detected errors and supported features of the Miri test suits `/fail/dangling` (Left) and `/pass/` (right) when monomorphizing.

5.4.3 Interpretation of Results

Miri currently, performs better than our analysis in the types of Rust features it supports. Additionally, it can detect Rust specific types of undefined behavior currently not supported by our analysis. A limitation of evaluating against the Miri test suite is that it inherently biases results toward features Miri supports. In particular, because Miri is a dynamic analyzer, its tests only surface errors along executed code paths, errors in unexecuted functions would go undetected. However, the Miri test bench is a good use case for empieracly showing that our analysis does not produce false positives. **Answer to RQ3:** *We have shown that our analysis can detect some of the classical memory errors in the Miri test suit but not yet Rust specific type of undefined behavior. Additionally, it did not produce any false positives on the safe dataset.*

5.5 Discussion

Even though currently only a subset of the Rust features are supported, the testing on both our created dataset and that of Miri has shown that our analysis already can detect undefined behavior of various kind of classical memory errors, and it has empirically shown that our analysis does not report false-positives on our own test suit and on those of Miri.

While we did not find any occurrences of false-positives during testing the code on the synthetic and Miri dataset there is one specific case where false-positives can occur due to the interaction of Infer with Charon. Pulse makes optimistic assumptions about calls to unknown functions. It will scramble the parts of the state reachable from the parameters of the call. In general, this helps avoid false positives but not in all cases. Take for example the code in Listing 5.6 where a false-positive is shown. Since in Charon functions from external libraries by default do not include the function body it can lead to false-positives in some cases. The code in Listing 5.7 will give a memory leak error unless the following flags are passed through Charon: `--include=core::mem::drop --monomorphize`.

Listing 5.6: False Postive in Pulse [29]

```
void false_positive(int *x) {
  unknown(x); // this sets *x to 5
  if (x != 5) {
    // unreachable
    int* p = NULL;
    *p = 42; // false positive reported here
  }
}
```

Listing 5.7: False Postive in Pulse in Rust

```
fn main() {
    let b = Box::new(10);
    core::mem::drop(b);
}
```

Chapter 6

Related work

In this chapter we will discuss the related work, and discuss some other tools used to detect undefined behavior in Rust.

6.1 Rustc Librarification Project Group

The representation from MIR can quite differ from version to version. Similar to Charon the *Rustc Librarification Project* intends to provide a stable Application Programming Interface (API) to interact with the Rust compiler. In contrast to Charon it is not a stand-alone Rust driver but instead designed as a thin wrapper over compiler internals to write your own rustc drivers. As such, it does not intend to clean up the representation of MIR as Charon does [14, 36].

6.2 Aeneas

AeneasVerif is the organization that maintains Charon. Besides Charon they also maintain several other projects for the verification of Rust programs. Eurydice is a Rust to C compiler. It's purpose is to provide a backwards-compatibility story as the verification ecosystem gradually transitions to Rust Aeneas is a verification toolchain for Rust programs. It translates Rust programs to several verification backends, including F*, Coq, HOL4 and Lean, which can be used to prove properties of Rust programs. Both projects use Charon for their translation from Rust [2].

6.3 Prusti

Prusti [3, 28, 11] is a software verification tool for verifying correctness of Rust programs. Prusti can detected correctness when provided with user defined annotations. Prusti encodes Rust and user annotations in Viper. Viper's logic is based on Dynamic Frames, which is a close relative of SL. Dynamic frames additionally allow the incorporation of heap dependent logic assertions. Assertions called accessibility predicates, written as `acc(e.f)`, are used to denote the exclusive field permission for the field f of the object denoted by e.

6.4 Kani

The Kani Rust Verifier is a bit-precise model checker for Rust. It can analyze Rust programs by creating test harnesses for function and providing it with a non-deterministic input. It can automatically checks for many kinds of undefined behavior and correctness properties by encoding them as Boolean Satisfiability Problem (SMT) and acSAT problems [24, 13].

6.5 Separation logic for Rust - Synthetic Ownership Logic

RusSOL is a tool for automatically synthesizing Rust programs from functional correctness specifications. It integrates Synthetic Ownership Logic (SOL) which is a variant of SL that incorporates Rust borrow and lifetime semantics. In SOL a heap is modeled as Locations that point to value and permissions: $Loc \mapsto (0, 1] \times Val$. A full permission $p = 1$ allows write access whereas a permission $0 < p < 1$ only allows read access. It additionally also models references by adding lifetime annotations to the heap representation. Variable of a reference type `type &'a mut T` are represented as a special kind of binding: $y \mapsto^{'a} T$ where 'a denotes the lifetime reference. Then every value in the heap is annotated with a blocking set that indicates the lifetimes of references that have borrowed from this variable [11].

6.6 Miri

Miri is an undefined-behaviour detection tool that can be used to check whether code written in unsafe Rust is actually safe. It is a dynamic tool which works at runtime in contrast to the borrow checker that works on compile time. Miri runs the code on a test suite to check whether it violates Rust safety guarantees. However, since Miri is dynamic, it will only check correctness of the parts of a program that are actually executed by the test suite [42]. Miri is maintained by the Rust-lang organization and is used as part of their CI.

6.7 Soteria

Soteria, is an OCaml library for writing Symbolic Execution (SE) engines in a functional style. It enables developers to construct SE engines that operate directly over source-language semantics, offering configurability, compositional reasoning, and ease of implementation. Two such SE engines are: the first Rust SE engine supporting Tree Borrows and Soteria^{rust} and Soteria^{C} a compositional SE engine for C. Soteria^{rust} offers compatibility with Kani's proof harnesses. Support for adding compositional analysis to Rust is currently a work in progress [5, 38].

Chapter 7

Conclusion

In this thesis we have shown how ISL can be used to detect memory errors. We have identified the fragment of memory bugs in Rust that can be detected using current ISL, and extended Infer with Rust support. While for now, only a subset of Rust features are supported we can already detect various kind of classical memory errors, and empirically showed that our analysis did not result in any false positives.

7.1 Future Work

To realize the full potential for detecting undefined behavior, first, we have to expand the translation to support a larger subset of Rust features, and, secondly, extend Infer to support analysis for Rust specific undefined behavior.

7.1.1 Translation

Our current translation already has support for important Rust features. Beside the basic operations such as control flow and binary/unary operations we support the use of raw pointers and references, although it currently does not make a distinction between the two. It supports boxes for primitive types, basic data-structures such as structs, and tuples. Additionaly, it has rudimentary support for enum types and error handling.

As was shown in Section 5.2, when using monomorphized functions we can translate more functions than without. However, being able to analysis functions with is useful when analyzing data structures and methods in, for example, a library.

Currently we have special handling for Box types, to support other special pointer types such as Rc, Arc, support need to be added for the UnsafeCell Primitive in Rust. In general more support need to be added for more compiler intrinsics including, but not limited to, functions such as `const fn size_of<T>()` and `Global` allocator functions as `exchange_malloc`.

`dyn` traits types are currently not supported, and monomorphized versions of `dyn` traits are currently still a work in progress in Charon.

One big feature of rust currently not supported by our translation are slices. Adding support for the slice type would be to treat the slice as a fat pointer, for example by treating it as a struct with a pointer field and size metadata. However, at the ULLBC level creating a slice is not a built-in operation, but a function call of a trait method from `Index`. This call goes several layers deep. It handles bound checking, error handling, dynamic trait resolution, and calls to compiler intrinsic, which would need to be supported before the slice expression would be supported. Boxes are currently modeled for primitive types, this can be expanded to when more compiler intrinsics are implemented to support more types in general.

7.1.2 Analysis

In Section 5.3 we have shown a few cases of undefined behavior in Rust currently not detectable by our analysis. To detect some of these kinds of undefined behavior, Infer's ISL would need to be expanded with support for stack borrow semantics [19] or tree borrow semantics [44].

Bibliography

[1] *a-proactive-approach-to-more-secure-code*. July 16, 2019. URL: https://www.microsoft.com/en-us/msrc/blog/2019/07/a-proactive-approach-to-more-secure-code (visited on 12/16/2025).

[2] a Verification Framework for Rust Aeneas. *Aeneas, a Verification Framework for Rust*. Aeneas, a Verification Framework for Rust. URL: https://aeneasverif.github.io/ (visited on 03/27/2026).

[3] Vytautas Astrauskas et al. "The Prusti Project: Formal Verification for Rust". In: *NASA Formal Methods*. Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Vol. 13260. Cham: Springer International Publishing, 2022, pp. 88–108. ISBN: 978-3-031-06772-3 978-3-031-06773-0. DOI: 10.1007/978-3-031-06773-0_5. URL: https://link.springer.com/10.1007/978-3-031-06773-0_5 (visited on 04/28/2025).

[4] Sacha-Élie Ayoun, Opale Sjöstedt, and Azalea Raad. *Soteria: Efficient Symbolic Execution as a Functional Library*. Nov. 14, 2025. DOI: 10.48550/arXiv.2511.08729. arXiv: 2511.08729 [cs]. URL: http://arxiv.org/abs/2511.08729 (visited on 11/19/2025). Prepublished.

[5] Sacha-Élie Ayoun, Opale Sjöstedt, and Azalea Raad. *Soteria: Efficient Symbolic Execution as a Functional Library*. Nov. 24, 2025. DOI: 10.48550/arXiv.2511.08729. arXiv: 2511.08729 [cs]. URL: http://arxiv.org/abs/2511.08729 (visited on 03/27/2026). Prepublished.

[6] Yechan Bae et al. "Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. New York, NY, USA: Association for Computing Machinery, Oct. 26, 2021, pp. 84–99. ISBN: 978-1-4503-8709-5. DOI: 10.1145/3477132.3483570. URL: https://dl.acm.org/doi/10.1145/3477132.3483570 (visited on 01/08/2026).

[7] *Behavior considered undefined - The Rust Reference*. URL: https://doc.rust-lang.org/reference/behavior-considered-undefined.html (visited on 03/25/2026).

[8] Cristiano Calcagno et al. "Compositional Shape Analysis by means of Bi-Abduction". In: (2009).

[9] *charon_0_1_174*. GitHub. URL: https://github.com/AeneasVerif/charon/commit/8f77f850bdb2da62cd9189d8438322d7c536dada (visited on 03/19/2026).

[10] *Commits · ArjanSeijs/infer*. GitHub. URL: https://github.com/ArjanSeijs/infer (visited on 03/19/2026).

[11] Jonáš Fiala et al. "Leveraging Rust Types for Program Synthesis". In: *Reproduction Package for Article "Leveraging Rust Types for Program Synthesis"* 7 (PLDI June 6, 2023), 164:1414–164:1437. DOI: 10.1145/3591278. URL: https://dl.acm.org/doi/10.1145/3591278 (visited on 04/09/2025).

[12] *Garbage Collector Implementation*. Oracle Help Center. URL: https://docs.oracle.com/en/java/javase/24/gctuning/garbage-collector-implementation.html#GUID-71D796B3-CBAB-4D80-B5C3-2620E45F6E5D (visited on 11/25/2025).

[13] *Getting started - The Kani Rust Verifier*. URL: https://model-checking.github.io/kani/getting-started.html (visited on 03/27/2026).

[14] Son Ho et al. "Charon: An Analysis Framework for Rust". In: *Computer Aided Verification*. Ed. by Ruzica Piskac and Zvonimir Rakamarić. Vol. 15934. Cham: Springer Nature Switzerland, 2025, pp. 377–391. ISBN: 978-3-031-98684-0 978-3-031-98685-7. DOI: 10.1007/978-3-031-98685-7_18. URL: https://link.springer.com/10.1007/978-3-031-98685-7_18 (visited on 12/03/2025).

[15] *Introduction - Rust By Example*. URL: https://doc.rust-lang.org/stable/rust-by-example/ (visited on 03/10/2026).

[16] Ralf Jung. "Understanding and evolving the Rust programming language". doctoralThesis. Saarländische Universitäts- und Landesbibliothek, 2020. DOI: 10.22028/D291-31946. URL: https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647 (visited on 02/05/2026).

[17] Ralf Jung et al. "Miri: Practical Undefined Behavior Detection for Rust". In: 10 ().

[18] Ralf Jung et al. "RustBelt: securing the foundations of the Rust programming language". In: *Proceedings of the ACM on Programming Languages* 2 (POPL Jan. 2018), pp. 1–34. ISSN: 2475-1421. DOI: 10.1145/3158154. URL: https://dl.acm.org/doi/10.1145/3158154 (visited on 06/03/2025).

[19] Ralf Jung et al. "Stacked borrows: an aliasing model for Rust". In: *Proceedings of the ACM on Programming Languages* 4 (POPL Jan. 2020), pp. 1–32. ISSN: 2475-1421. DOI: 10.1145/3371109. URL: https://dl.acm.org/doi/10.1145/3371109 (visited on 07/24/2025).

[20] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* International Symposium on Code Generation and Optimization, 2004. CGO 2004. Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665. URL: https://ieeexplore.ieee.org/abstract/document/1281665 (visited on 02/10/2026).

[21] Andrea Lattuada et al. "Verus: Verifying Rust Programs using Linear Ghost Types". In: *Proceedings of the ACM on Programming Languages* 7 (OOPSLA1 Apr. 6, 2023), pp. 286–315. ISSN: 2475-1421. DOI: 10.1145/3586037. URL: https://dl.acm.org/doi/10.1145/3586037 (visited on 01/07/2026).

[22] Nicholas D. Matsakis and Felix S. Klock. "The rust language". In: *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*. HILT '14. New York, NY, USA: Association for Computing Machinery, Oct. 18, 2014, pp. 103–104. ISBN: 978-1-4503-3217-0. DOI: 10.1145/2663171.2663188. URL: https://dl.acm.org/doi/10.1145/2663171.2663188 (visited on 06/03/2025).

[23] Nicholas D. Matsakis and Felix S. Klock. "The rust language". In: *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*. HILT '14: High Integrity Language Technology ACM SIGAda Annual Conference. Portland Oregon USA: ACM, Oct. 18, 2014, pp. 103–104. ISBN: 978-1-4503-3217-0. DOI: 10.1145/2663171.2663188. URL: https://dl.acm.org/doi/10.1145/2663171.2663188 (visited on 12/17/2025).

[24] *model-checking/kani*. model-checking, Mar. 26, 2026. URL: https://github.com/model-checking/kani (visited on 03/27/2026).

[25] Peter W. O'Hearn. "Incorrectness logic". In: *Proceedings of the ACM on Programming Languages* 4 (POPL Jan. 2020), pp. 1–32. ISSN: 2475-1421. DOI: 10.1145/3371078. URL: https://dl.acm.org/doi/10.1145/3371078 (visited on 03/07/2025).

[26] *PLDI'23 Infer Sunday Magnolia5*. URL: https://www.youtube.com/watch?v=_wPTcXuAWPY (visited on 01/14/2026).

[27] *Prologue - Rust Compiler Development Guide*. URL: https://rustc-dev-guide.rust-lang.org/part-3-intro.html (visited on 02/09/2026).

[28] *Prusti*. Programming Methodology Group. URL: https://www.pm.inf.ethz.ch/research/prusti.html (visited on 04/09/2025).

[29] *Pulse | Infer*. URL: https://fbinfer.com/docs/checker-pulse/ (visited on 03/07/2025).

[30] Azalea Raad et al. "Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic". In: *Computer Aided Verification*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Cham: Springer International Publishing, 2020, pp. 225–252. ISBN: 978-3-030-53290-1 978-3-030-53291-8. DOI: 10.1007/978-3-030-53291-8_14. URL: http://link.springer.com/10.1007/978-3-030-53291-8_14 (visited on 03/07/2025).

[31] J.C. Reynolds. "Separation logic: a logic for shared mutable data structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 17th Annual IEEE Symposium on Logic in Computer Science. July 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817. URL: https://ieeexplore.ieee.org/document/1029817/ (visited on 05/28/2025).

[32] *Rust for Linux*. URL: https://rust-for-linux.com (visited on 03/06/2025).

[33] *Rust Programming Language*. URL: https://rust-lang.org/ (visited on 11/19/2025).

[34] *rust-lang/miri*. The Rust Programming Language, June 4, 2025. URL: https://github.com/rust-lang/miri (visited on 06/04/2025).

[35] *rustc-efc9e1b50*. GitHub. URL: https://github.com/rust-lang/rust/commit/efc9e1b50cbf2cede7ebe25f0 (visited on 03/20/2026).

[36] *rustc_public - Rust*. URL: https://doc.rust-lang.org/nightly/nightly-rustc/rustc_public/index.html (visited on 03/27/2026).

[37] *Separation logic and bi-abduction | Infer*. URL: https://fbinfer.com/docs/separation-logic-and-bi-abduction/ (visited on 02/23/2025).

[38] *Soteria - Sound Static Analysis for the Masses*. Soteria. URL: https://soteria-tools.com/ (visited on 03/27/2026).

[39] *The borrow checker - Rust Compiler Development Guide*. URL: https://rustc-dev-guide.rust-lang.org/borrow-check.html (visited on 02/04/2026).

[40] *The MIR (Mid-level IR) - Rust Compiler Development Guide*. URL: https://rustc-dev-guide.rust-lang.org/mir/index.html#mir-data-types (visited on 12/03/2025).

[41] *The Rust Programming Language - The Rust Programming Language*. URL: https://doc.rust-lang.org/book/title-page.html (visited on 03/07/2025).

[42] *Unsafe Rust - The Rust Programming Language*. URL: https://doc.rust-lang.org/book/ch20-01-unsafe-rust.html (visited on 03/07/2025).

[43] Alexa VanHattum et al. "Verifying dynamic trait objects in rust". In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '22. New York, NY, USA: Association for Computing Machinery, Oct. 17, 2022, pp. 321–330. ISBN: 978-1-4503-9226-6. DOI: 10.1145/3510457.3513031. URL: https://dl.acm.org/doi/10.1145/3510457.3513031 (visited on 12/19/2025).

[44] Neven Villani et al. “Tree Borrows”. In: *Tree Borrows – Artifact* 9 (PLDI June 13, 2025), 188:1019–188:1042. DOI: 10.1145/3735592. URL: https://dl.acm.org/doi/10.1145/3735592 (visited on 11/19/2025).

Acronyms

SL separation logic

IL incorrectness logic

ISL incorrectness separation logic

SIL smallfoot intermediate language

SOL Synthetic Ownership Logic

CI Continuous Integration

IR Intermediate Representation

CFG Control flow Graph

MIR Mid-Level Intermediate Representation

ULLBC Unstructured Low-Level Borrow Calculus

AST Abstract Syntax tree

HIR High-Level Intermediate Representation

FFI Foreign Function Interface

THIR Typed High Level Intermediate Representation

API Application Programming Interface

SAT Satisfiability Modulo Theories

SMT Boolean Satisfiability Problem

SE Symbolic Execution