

Contents

Contents	1
1 Introduction	2
1.1 The Rust Ownership and Borrow System	2
1.1.1 Ownership	2
Resource Handling Strategies	3
RAII and Drop Responsibilities	4
Move Semantics	5
Clone	5
Copy	6
ManuallyDrop	6
1.1.2 Borrowing	7
The Owner of a Borrowed Value	8
Lexical Lifetimes and Lifetime Analysis	8
Interior Mutability and Access Guards	9
Returning references and Borrow-through	9
Lifetime Polymorphism	10
1.1.3 Reference Conversions	11
AsRef	11
Deref	12
Borrow	12
Mutable reference conversions	12
1.1.4 Non-Lexical Lifetimes	12
1.1.5 Reborrows and Two-Phase Borrowing	13
Reborrows	15
Two-Phase Borrowing	15
Generalized Two-Phase Borrow (TPB)	16
1.1.6 Loans and Regions	17
Borrow Errors	17
Classic Non-Lexical Lifetimes (NLL)	18
Polonius	18
Self-referential Structs	20
Polonius the Crab	20
Cyclone	20
1.2 Contracts and Refinement Types	20
1.2.1 Contract Programming	21
Assertions	24
Integrated Contract Programming	25
loop invariants/conditions	25
Higher-order contracts	25
1.2.2 Refinement Types	25
Refinement Types in Functions	27
1.2.3 Liquid Types and SMT Solving	27
1.2.4 Hybrid Contract Checking	27
1.3 Other Extensions to a Type System	28
1.3.1 Totality and Termination Metrics	28
1.3.2 (Other) Substructural Types	28

1.3.3	Purity and (algebraic) Effects	28
2	Formal Semantics for Combining OBS and Refinement Types	29
2.1	Kinds of Formal Semantics	29
2.2	Survey of Formal Semantics for Rust	30
2.3	Discussion	31

1 Introduction

The goal of this thesis is to combine the concept of an Ownership and Borrow System that the Rust programming language supports with Refinement Types as found in some functional languages like ATS and Liquid Haskell and theorem provers like Lean.

In this chapter we will explore these concepts before we try to unify them in the rest of the thesis. In section 1.1, we look at how Rust’s type system implements ownership and borrowing, section 1.2 covers refinement types and related concepts, and section 1.3 briefly discusses further ways to enhance a type system for a safer language.

1.1 The Rust Ownership and Borrow System

Resource management is hard, especially memory management. This becomes apparent when looking at a study by Microsoft [Thou9] that found out that the vast majority of security related bugs in their code was due to corrupted memory.

Therefore, most programming languages automate memory management with garbage collection (GC). In a garbage collected language, no memory corruption can happen. Of course, it comes with a drawback: The programmer has to give up control over when and how GC is run, which is undesirable for performance critical software.

A different approach is using Resource Acquisition is Initialization (RAII) like C++ does for some of its types and which does static analysis on the code to insert cleanup code at the correct places. Rust is a new language that embraces this kind of memory management in its Ownership and Borrow System (OBS). In this section we will explore the intricacies of OBS as Rust uses it.

1.1.1 Ownership

In most programming languages, one can freely create and delete resources. For example, in C the programmer allocates memory on the heap with `malloc` and frees the memory when they don’t need the memory anymore using `free`.

```
// Allocate space for an `int`.
int *ptr = (int*) malloc(sizeof(int));

// ...

// Free the memory.
free(ptr);
```

Listing 1.1: Memory management in C

Even in languages without manual memory management, the programmer still must manage other resources. For example, to work with a file in Python one could write:

```
# Open the file and store the file handle in a variable.
file = open("path/to/some.file", "r")

# ...

# Close the file
file.close()
```

Listing 1.2: Manual resource management in Python

Resource Handling Strategies This manual resource management comes with some difficulties. The programmer has to watch out to always free the resources after using them, because otherwise the program could leak memory or unnecessarily inhibit other processes from accessing the resources. On the other hand, one may also not free the resources too early, which would lead to a use after free, and also not free resources multiple times. This can be very tricky in complex programs where resources are shared between threads or over API boundaries.

That is why several programming languages have ways to make handling resources easier. Python has context managers that close resources automatically.

```
with open("path/to/some.file", "r") as file:
    #...

# At this point `file` is closed.
```

Listing 1.3: Context managers in Python

This code is equivalent to the previous one, but even better: It also closes the file in case of an exception. Programmers can also define custom context managers for their own resources.

Other languages have different strategies. Go supports the **defer** statement which takes an operation that is not to be run immediately but at the end of the scope.

```

{ // Block starts here.
  file, err := os.Open("path/to/some.file")
  defer file.Close()

  // ...

} // Block ends here; run the deferred close file

```

Listing 1.4: The **defer** statement in Go

These prevent double frees and use after free errors, but they can still be forgotten. A programmer who wants to use a resource must be aware that it has to be cleaned up in the end and changes in one line of the code have to be mirrored in the other. It is not as bad as in manual resource management because the programmer has to change only one additional line and that line is usually close.

RAII and Drop Responsibilities A third variant is *Resource Acquisition is Initialization* (RAII), also called *Space Bound Resource Management* (SBRM), used by C++ and Rust. This ensures that the destructor of an object is run when it goes out of scope, which will clean up automatically. An example is the **std::unique_ptr** in C++. The pointer *owns* the memory it points to, meaning when it is created, it automatically allocates memory on the heap and when it goes out of scope, it frees the memory.

```

{ // Block starts here.
  std::unique_ptr<int> ptr = std::make_unique<int>(42);

  // ...

} // Block ends here; run the destructor and free the memory

```

Listing 1.5: RAII in C++

RAII ensures that resources are always freed when they are not live (that is, may not be used) anymore. The programmer can't forget to free the resources nor cause use after free or double free. While in C++ only certain types conform to RAII, in Rust every value is owned by a variable. For example, the Rust analog to an owning pointer is **Box**.

```

{ // Block starts here.
  let ptr = Box::new(42)

  // ...

} // Block ends here; drop value and free memory

```

Listing 1.6: Heap allocation in Rust

As soon as a variable falls out of scope, it is *dropped*, meaning all associated resources are freed. It is said that a variable has a *drop responsibility*. A programmer can define custom drop behavior for their own resources by implementing the **Drop** trait for their type, but normally the compiler does it for us.

Move Semantics RAII comes with a few disadvantages, though. Since a value is dropped when its owner goes out of scope, there must always be exactly one owner for every value. This means that the compiler will transfer ownership sometimes, like in the following example.

```
fn print_value(ptr: Box<i32>) {
    println!("{}", *ptr);
}

fn main() {
    let mut ptr = Box::new(42);

    print_value(ptr);

    *ptr += 1;
}
```

Listing 1.7: Ownership transfer

We create a pointer and use it as an argument to a function that prints the pointee's value. After that we increment the value. But if we try to compile this, we get an error:

```
error[E0382]: use of moved value: `*ptr`
```

Because we supplied **ptr** as an argument to **print_value**, it took ownership of that value, and now has the drop responsibility. As soon as the call to **print_value** is finished, the memory is freed and accessing it afterwards is a use after free. We say that the value was *moved* out of **ptr** and into **print_value**.

Clone We can still make it work by *cloning* **ptr** and supply the clone as an argument. **ptr.clone()** creates a new **Box**, allocates new memory, and initializes it with the same value as **ptr**'s:

```

fn print_value(ptr: Box<i32>) {
    println!("{}", *ptr);
}

fn main() {
    let ptr = Box::new(42);

    print_value(ptr.clone());

    *ptr += 1;
}

```

Listing 1.8: Usage of **Clone**

Now everything works but keep in mind that the clone is completely independent from the original value. If we changed the value of the clone in `print_value`, it would not be visible to the outside.

Copy We previously noted that ownership, and therefore move semantics, apply to every value in Rust. But if we change the code to use an `i32` directly instead of putting it on the heap, we don't need to clone anything:

```

fn print_value(number: i32) {
    println!("{}", number);
}

fn main() {
    let mut number = 42;

    print_value(number);

    number += 1;
}

```

Listing 1.9: Number types are **Copy**

This compiles and works. That is not because `i32` breaks move semantics but because the type implements the **Copy** trait. Normally, if a value is moved, the physical bits making up that value are moved to a new location, e. g. the new stack frame, and are not available at the previous position anymore. If a type implements **Copy**, the bit pattern is instead copied over and retained. Because `i32` doesn't allocate any heap or handles any resources, such a copy is valid. A **Box** is not **Copy**, because then two owners for the same resource would exist which would violate drop responsibility. If we want to duplicate a **Box**, we need to allocate new memory on the heap and initialize it properly, which is what `clone` does.

ManuallyDrop Sometimes we don't want RAII to happen, we want to free resources ourselves. If that is the case, we can wrap a value in a **ManuallyDrop**

which tells the compiler to not drop it for us. This is a general trend in Rust: The correct way should be the easiest to do, and all potentially unsafe constructs are opt-in. More information on `ManuallyDrop` can be found in the documentation for the type.¹

1.1.2 Borrowing

As we have seen, ownership and move semantics ensure memory safety. But they are not easy to use, `clone`ing data has a runtime overhead, and many correct programs are rejected. For example, Listing 1.7 describes a program that would work if it were not for move semantics.

That is where references and borrowing come in. Instead of taking ownership of a value, a function can only borrow it through a reference. Then the drop responsibility stays with the caller. References, of course, can not be used for everything, but for our case it is sufficient. We mark the argument to `print_value` as a reference using `&`, and creating a reference from a value works the same.

```
fn print_value(ptr: &Box<i32>) {
    println!("{}", *ptr);
}

fn main() {
    let mut ptr = Box::new(42);

    print_value(&ptr);

    *ptr += 1;
}
```

Listing 1.10: Borrow using references to prevent a move

Listing 1.10 compiles without an error and does what we would expect. There are two kinds of references. We just looked at *shared* or *immutable references*. The other kind is *exclusive* or *mutable references*, and they are denoted with `&mut`. The different kinds of references have a different semantics attached to them. While both are used to access values without taking ownership, there are specific rules for the creation and guarantees associated with each:

- A shared reference to a value can always be created, as long as there is no exclusive reference to the same value. It is not possible to mutate the pointee through a shared reference (which is why it is also called immutable).
- An exclusive reference can only be created if there is no other reference to the value at all and the referenced value is declared mutable. An exclusive reference can do everything the owner can, except dropping.

¹<https://doc.rust-lang.org/stable/std/mem/struct.ManuallyDrop.html>

The part of the compiler that enforces these rules is the *borrow checker*. The reasoning behind this is again resource safety, namely preventing *unguarded mutable aliasing*. Having multiple readers of the same data doesn't cause issues, as long as the data cannot be mutated. Mutating data is fine, as long as no one else can read and/or mutate the data at the same time. Using these two kinds of references enforces, at compile time, that we will always stay on the happy path. But there are some programs that are correct even though they violate these rules. We will be concerned with extending the borrow checker to accept more correct programs in later sections.

The Owner of a Borrowed Value References have to agree to these rules, but the owner has to as well. While shared references exist, the owner may not mutate, e. g. `let mut x = 42; let xref = &x; x += 1` is forbidden. Similarly, the owner can't access a value at all as long as there is an exclusive reference around.

This doesn't not work, I have to think about this some more. I want something to throw Error E0507.

In particular, a value may not be moved, as long as there is a reference to it because that would create a dangling pointer. This means that `let y = &x; let z = *y;` doesn't work.

Lexical Lifetimes and Lifetime Analysis The parts of the program in which a reference is valid is called its *lifetime*. The borrow checker keeps track of all references' lifetimes and their so-called *provenances*, that is, who the original owner of the referenced value is. References are values like all others and so they are dropped at the end of the scope they are defined in. In other words: Their lifetime starts at their creation and ends at the end of the scope. This is called a *lexical lifetime*. Therefore the code in Listing 1.11 does not compile.

```
let mut x = 0;
{
    let xref = &x; // `xref` is created
    println!("{xref}");
    x += 1; // error
} // `xref` is dropped
x += 1; // OK, no references exist anymore
```

Listing 1.11: Lexical Lifetimes prevent usage

The program in Listing 1.11 is rejected by our borrow checker since it registers a mutation to `x` while a reference to it still exists. Only after the block ends, the borrow is returned to the owner and it can be used again. Similarly, we would not be able to create an exclusive reference while a shared one exists and vice versa.

But we can see that the program is correct since `xref` is never accessed after `x` is changed and we can save the code by introducing an additional scope.


```

let mut x = 0;
{
    {
        let xref = &x; // `xref` is created
        println!("{xref}");
    } // `xref` is dropped
    x += 1;
}
x += 1;

```

Listing 1.12: More scopes make the code check

This is tedious, though, and we would like the borrow checker to recognize patterns like this automatically. Luckily, Rust implements an improved borrow checker since version 1.31. Because it is not based on lexical scope for lifetime analysis anymore, these new lifetimes are called *Non-Lexical Lifetimes* (NLL). We will take a closer look at them in subsection 1.1.4.

Interior Mutability and Access Guards The borrow rules are enforced at compile time, but sometimes the compiler can't know if a piece of code follows the rules. Because of this, there is a runtime-checked version of reference with `RefCell` which causes a panic if the borrow rules are broken. Also, sometimes we need access to a resource from multiple places at the same time, for example when sharing data between threads. For this, Rust provides the `Mutex` container type. References to a mutex can be shared freely, but to change the value in the container, one has to acquire a lock, therefore making the access *guarded*. While the mutex is locked, no other thread can access the data at all, as if the thread held a `&mut`. Alternatively, `RwLock` can give out both read and write locks which have behavior analogous to `&` and `&mut`, respectively. There are more constructs for similar use cases in the standard library, like `Arc` and `Cow`.

These types are implemented using the `Cell` type which can change a value even if it is not marked as mutable. This is called *interior mutability* and uses *unsafe Rust* internally, a superset of standard, safe Rust. The main feature of unsafe Rust is access to so-called *raw pointers*. Raw pointers are, like pointers in C, not borrow checked by the compiler. Since borrow checking is undecidable, the compiler sometimes can't prove that, for example, two references don't alias. In these cases, the programmer can step in, prove the non-aliasing manually, implement a feature using raw pointers and provide a safe abstraction for consumers of the code.

Pointers
to Strict
Provenance,
Miri, ...?

Returning references and Borrow-through Functions can receive references as arguments, but they can also return references. One has to be a bit careful when doing this, though, since all resources created in the scope of a function are freed as soon as the function returns. Consider the following:

```
fn to_ref(number: i32) -> &i32 {
    &number
}
```

Listing 1.13: Try to return a reference

This fails because `number` is owned by `to_ref` and is dropped as soon as the function returns. The reference would already be invalid when the caller gets access to it. But if the function takes a reference as an argument, it can pass another reference back to the caller. This is called a *borrow-through* and looks like the following:

```
fn to_ref(number: &i32) -> &i32 {
    &number
}
```

Listing 1.14: Borrow-through

Now Rust can couple both references to each other. The returned reference has the same provenance as the input one, and the borrow checker can check that the input reference's lifetime is at least as long as the returned one.

Lifetime Polymorphism Consider a function that takes two references and returns one reference itself. Imagine for example, we have two slices and want to create an iterator over all elements of both slices. Then we could try to write a function like in Listing 1.15. The `impl Iterator<Item=&T>` is a so-called existential type. It just means that we don't care about what type exactly it is as long as it is an iterator.

```
fn both(first: &[i32], second: &[i32])
    -> impl Iterator<Item=&i32>
{
    first.iter().chain(second.iter())
}
```

Listing 1.15: Iterator over two slices

This does not work however.

error[E0106]: missing lifetime specifier

The compiler is confused because it does not know which provenance and lifetime to assign to the returned references. We as programmers can see that the return value depends on both input references, so we can help Rust by providing lifetime hints.²

²Actually, the compiler could infer these lifetimes but it would rely on global program analysis to do so. It is also possible to infer function types, but Rust chose to always be explicit about the types and lifetimes you use and have them be explicit parts of the API of a function.

```
fn both<'a>(first: &'a [i32], second: &'a [i32])
    -> impl Iterator<Item=&'a i32>
{
    first.iter().chain(second.iter())
}
```

Listing 1.16: Iterator over two slices with lifetime parameters

'a is a *polymorphic lifetime parameter*. We tell Rust that the output relies on both inputs being valid for at least as long as itself. If the output relied on only one of the arguments, we would mark only one of them with a lifetime specifier. This would for example be the case if we want to search for an element in a slice and, if it exists, return a mutable reference to that element. We implement such a function in Listing 1.17.

```
fn find_mut<'a>(haystack: &'a mut [i32], needle: &i32)
    -> Option<&'a mut i32>
{
    haystack.iter_mut().find(|x| **x == *needle)
}
```

Listing 1.17: The output lifetime depends only on one of the inputs

1.1.3 Reference Conversions

Sometimes, we have a reference to one type but need a reference to another, similar type. For example, a vector of some type `Vec<T>` is conceptually the same as a slice of that same type `[T]`, except that a `Vec` can grow and shrink and a slice can not. This means that all functions which operate on slices should also work with vectors, and in fact there is a library method `Vec::as_slice` that takes a reference to a vector and provides a reference to a slice. Similarly there is `String::as_str` which transforms a `&String` into a `&str`.

AsRef Generally, there are many types that can act as a substitute for another. The common interface for this behavior is the `AsRef` trait. There can be many implementations of this trait for a given type. for example, `String` can stand in for `str`, `[u8]`, `OsStr` and `Path`. Every time a reference to one of these types are needed, we can use a `String` instead, if we first call `as_ref` on it:

```
fn needs_bytes(x: &[u8]) { /*...*/ }

// ...

let s = String::from("Hello Bytes");
needs_bytes(s.as_ref());
```

Listing 1.18: Use `String` and `as_ref` in place of `[u8]`

Deref Always calling `as_ref` is a bit cumbersome, especially if it is clear which type the target should be. For example, if we have a `Box<T>` there is really only one reasonable type we want to get out of it, namely a `&T`. The trait **Deref** can be used for that. It specifies a single type which Rust automatically converts into if it is needed. For example, `String` has `str` as its **Deref** target which allows us to omit the `as_ref` in that case and leave the conversion implicit:

```
fn needs_str(x: &str) { /*...*/ }

// ...

let s = String::from("Hello Bytes");
needs_bytes(s);
```

Listing 1.19: Use `String` in place of `[u8]`

We couldn't do the same in Listing 1.18 because `[u8]` is not **Deref** target of `String`.

Borrow Sometimes we want to express an even stronger connection between two types. For example, a `HashMap` needs to take ownership of its entries but we want to be able to do a lookup even if we only have a reference. But that requires that the owned and the borrowed value behave exactly the same. This is what the **Borrow** trait signals. In particular, `x.borrow() == y.borrow()` if and only if `x == y` and `x.hash() == x.borrow().hash()`.

Mutable reference conversions All three of the aforementioned traits work for shared references only, but their variants `AsMut`, `DerefMut` and `BorrowMut` all take and provide an exclusive reference.

1.1.4 Non-Lexical Lifetimes

In Listing 1.11 we saw how lexical lifetimes stood in our way and we had to manually wrap a reference in an additional block just to make our correct program pass the borrow checker. This was because the lifetime of any value is bound to its scope, at the end of which it is dropped. Non-Lexical Lifetimes are a different approach to lifetime analysis. A reference is live for as long as it may be used later. The borrow checker determines the points of the program in which the reference is live by building a *control-flow graph* (CFG). All nodes of the CFG that can be reached from the point of creation until the last use of the reference are where it is live. Consider the program in Listing 1.20.

```
let mut x = 0;
let xref = &x;
println!("{xref}");
x += 1;
```

Listing 1.20: Simple NLL example

Then the (simplified) CFG looks something like Figure 1. It is quite boring because the control flow is linear. The nodes with thick borders are where **xref** is live. The first node is the one in which the reference is created and the second one is where it is last used. Because **x** is not modified in that section of the CFG, the borrow checker doesn't complain.

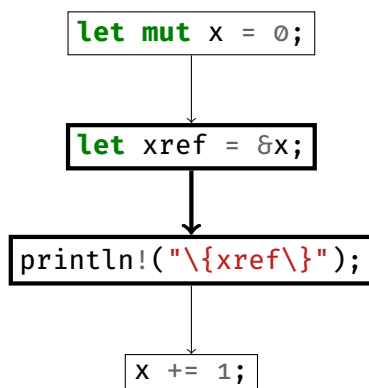


Figure 1: Control-flow diagram for Listing 1.20

Consider a more complicated example that includes branching in Listing 1.21.

```

let mut x = 0;
let xref = &x;
if xref != &0 {
    x += 1;
    println!("xref equals {xref}");
} else {
    println!("xref is 0");
}
x = 42;
  
```

Listing 1.21: NLL example with bracing

Now the CFG (Figure 2) splits up to accommodate both possible paths the program could take during execution. At the **if**, the graph splits into two, but in the last node, both paths join up again. Here we can see that in the **false** branch no problem occurs, but in the **true** branch we try to modify **x** even though **xref** is used later in the same path. Modifying **x** on the last line does not pose any problems since **xref** is not live anymore on any path that leads to this point.

1.1.5 Reborrows and Two-Phase Borrowing

NLL give us a lot more freedom when using references, but there are still programs that are clearly correct but don't pass the borrow checker, especially if exclusive references are in play. It is not possible to create a new reference

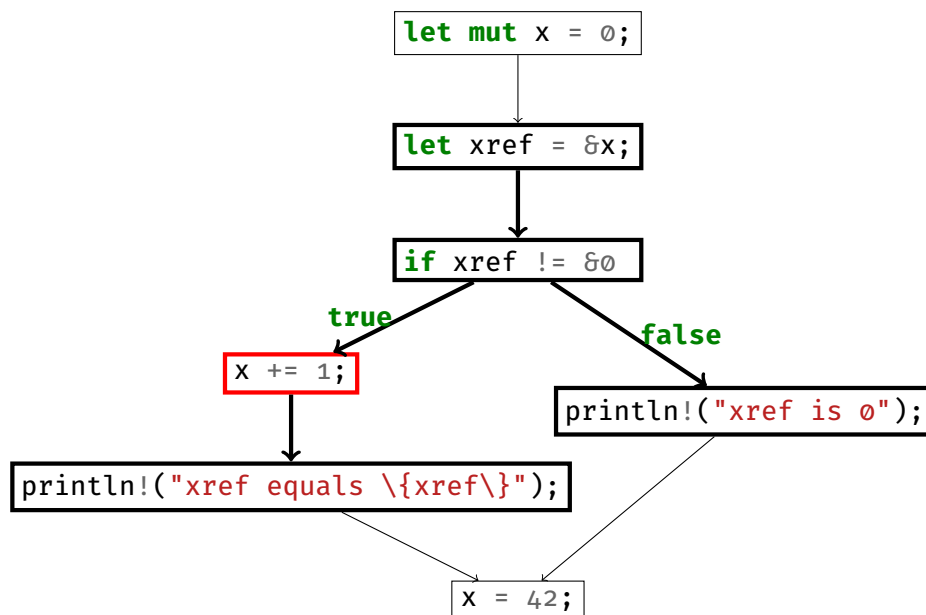


Figure 2: Control-flow diagram for Listing 1.21

of some provenance as long as an exclusive reference with the same provenance is live. This means the code in Listing 1.22 will not compile.

```

let mut x = 1;
let ref1 = &mut x;
let ref2 = &x; // Error: Cannot borrow `x`
println!("{ref2}");
*ref1 = 2;
  
```

Listing 1.22: Cannot create a shared reference while an exclusive one exists

But since **xs** is not live anymore when **xu** is used, aliasing is never occurring, and if we hadn't used an exclusive reference but modified the value directly, Rust would have been able to see this. Not being able to create references in this manner is a big thing. Consider the code in Listing 1.23. Here, we create a reference to an empty **Vec** and then use it to **push** a value. This moves the reference into the **push** method so that it is dropped when the method returns. It is no longer valid to use it again in the next line since it has already been dropped. But in fact this code compiles because Rust implicitly inserts a *reborrow* for us.

```

let mut v = Vec::new();
let vref = &mut v;
vref.push(1);
vref.last(); // Error: Use of moved value `vref`
  
```

Listing 1.23: Move a borrow into a function

Reborrows Let's get back to the example in Listing 1.22 for now. The compiler complains because we try to create two conflicting references with the same provenance. But we can tell the compiler to temporarily deactivate a reference by borrowing *through this reference*. This is done in Listing 1.24.

```
let mut x = 1;
let ref1 = &mut x;
let ref2 = &*ref1; // Reborrow
println!("{ref2}");
*ref1 = 2;
```

Listing 1.24: Reborrow through an exclusive reference

Now we can create and use the **ref2** as long as it is not live anymore when we use **ref1** again. If the last two lines were swapped, the borrow checker would correctly find the interleaved use and throw an error. It is possible to nest reborrows, as long as they are never used interleaved. These reborrows can be imagined as a stack: when creating a reborrow, the new reference is pushed to the stack, and when using a reference, the stack is popped until it is on the topmost position. Trying to access a reference that has already been popped indicates an interleaved use and is a borrow error. The owner of the value can then be thought of as the bottom element of the stack, and when it goes out of scope, the whole stack is unwound so that the **drop** function can access it. In fact, this is exactly how reborrows are modelled in Miri, which is an interpreter for Rusts *Mid-level Intermediate Representation* (MIR). [Miri](#) even has borrow stacks for raw pointers.

Source, e. g.
Rustonomi-
con

But why does the code in Listing 1.23 work? The compiler can insert borrows and reborrows in function calls, so-called *autorefs*. This is because it can be sure that the function returns before the reference is accessed again, meaning that the borrow stack is kept intact. In fact, if the compiler can't proof that the borrow stack is valid, it will complain. This can for example happen when using **std::thread::spawn** which creates a new thread. Because the child thread could outlive the main thread, Rust can't verify that all references are used in the correct order, in particular that the owner is not dropped too early.

Two-Phase Borrowing Reborrows won't solve every problem, though. Consider the code in Listing 1.25. Here we create a vector and then push its length onto it.

```
let mut v = Vec::new();
v.push(v.len());
```

Listing 1.25: Borrow twice in one method call

The problem lies in the second line. The first argument to **push** is an **&mut self**, and so the compiler implicitly borrows from **v**. But then another (shared) reference is created for the call to **len**. This is not allowed, and a reborrow doesn't help either since the **&mut self** is currently used. On

the other hand, it is clear that the call to `len` will definitely return before the exclusive reference is ever accessed. In fact, we can work around this problem if we save the result of `v.len()` in a temporary variable, as shown in Listing 1.26.

```
let mut v = Vec::new();
let vlen = v.len();
v.push(vlen);
```

Listing 1.26: Workaround for Listing 1.25

This pattern—calling a method with an exclusive reference but also using shared references in the arguments—is very common and the workaround is unwieldy. Therefore RFC 2025 [Mat17] introduces the concept of a TPB. In it, the lifetime of an exclusive reference is split up into two phases. During the *reservation phase*, it is treated like a shared reference, meaning more shared references can be created and reads are possible. The *activation* happens as soon as the reference is first used to perform a mutation. From this point on, it is treated as a full exclusive reference.

Generalized TPB Right now, a TPB may only happen in a few specific places, namely when calling a method with `&mut self` as first argument and with assignment operators (`+=`, `*=`, ...), and the other references can only be shared. There are several ways in which the requirements can be relaxed and which are discussed in the RFC and Issue #49434³.

Currently, the only place an exclusive reference is allowed is in `&mut self` position. This means that it is not possible to use an exclusive reference twice, and a reborrow is also not possible because the references have no names. One could relax this and allow exclusive references everywhere, and also mix shared and exclusive references. This poses the difficulty that a shared reference (or one during reservation) can observe change. Resolving this makes the model more complicated.

In a similar vein, TPB could be expanded to all function calls or even in inline code, as shown in Listing 1.27. Here, creating a shared reference is not allowed because an exclusive one already exists and is live. With generalized TPB it would be okay, though, since `xm` is only in reservation phase at that point.

```
let mut x = 1;
let xm = &mut x;
let xr = &x; // Error
*xm = 2;
```

Listing 1.27: Example of inline TPB

Even further, *discontinuous borrows* would make it possible that a borrow is only active at exactly the points at which it is used and deactivated in between.

³<https://github.com/rust-lang/rust/issues/49434>

Also, if the mutation capabilities of the reference is not used anymore, it could be *downgraded* to a shared reference: For example, a function could take an exclusive reference to a value but then returns only a shared one.

All of these changes would entail a big step up in complexity and surface area of the language without actually having much practical use since relevant situations are rare. They are discussed in detail in [Mat17].

1.1.6 Loans and Regions

Previously, we looked at the advantages of Non-Lexical Lifetimes over standard lexical ones. But lifetimes have a general defect and are supposed to be replaced by an approach using *regions* employed by the *Polonius borrow checker*, which is currently in an experimental stage. Here, we will explore how NLL and regions work under the hood, and what the advantage of using regions is.

Borrow Errors We need to define some vocabulary first: A *path* is an identifier like `x`, or is built from a path by a field access `x.f`, a dereference `*x`, or an index `x[i]`. Those can be freely combined, so that for example `(*x)[5].f` is a valid path.⁴

A *loan* is the result of a borrow expression. It consists of the path which is borrowed from and a *mode*, that is shared or exclusive.

A loan *L* is *violated* if its path *P* is accessed in an incompatible way, that is, *P* is mutated when the *L* is shared, or *P* is accessed at all when *L* is exclusive. Note that an access can also be *indirect* if *P* shows up somewhere in the expression. For example, `(*x)[5].f` accesses `(*x)[5]`, `*x`, and `x` indirectly. Note also that a loan to an index ignores the index variable, that is `x[5]` and `x[4]` produce the same loan to `x[_]`. This is because Rust can generally not know at compile time if two index operations alias. It means that it is impossible to have two exclusive references to different parts of a data structure like in Listing 1.28.

```
let mut v = vec![1, 2];
two_refs(&mut v[0], &mut v[1]);
```

Listing 1.28: Indexing twice into a vector is illegal

Now we can define when a borrow error should occur. There are three conditions which all have to be met:

1. A path *P* is accessed at some node *N* of the CFG,
2. accessing *P* at *N* would violate some loan *L*, and
3. *L* is live at *N*.

Different approaches to borrow checking only differ in determining when *L* is live. For example, with lexical lifetimes a loan is simply live from its creation until the end of the lexical scope. We are now prepared to dive into liveness analysis in NLL and Polonius.

⁴Paths have a rough equivalent in C and C++ lvalues.

Classic NLL Under NLL the liveness of a loan is derived from the lifetime of a reference. As discussed in subsection 1.1.4, a reference is live in a node of the CFG if it may be used later. This means that we walk forward along the CFG to determine the liveness of the reference and the corresponding loan is live exactly when the reference is. Crucially, if a function returns a reference, it is live for the whole body of the function.

```
fn first_or_insert(v: &mut Vec<i32>) -> &i32 {
  let fst = v.first();
  match fst {
    Some(x) => x,
    None => {v.push(1); &v[0]},
  }
}
```

lifetime subtyping and inferred lifetimes/constraints, “outlive” relationship, 'a : 'b

Listing 1.29: NLL reject correct program

Listing 1.29 shows an example. In the **Some** branch, **x** is returned from the function, which in turn depends on the reference produced by **fst**. Because **x** is returned from the function, it needs to be live at least until the end of the body of **first_or_insert**. But since **x** is derived from **fst**, that reference must outlive **x**, hence being live for the whole function body as well. In the **None** branch, an exclusive reference to **v** is created for the call to **push**. This produces an error because that node lies on a path between the creation of **fst** and the return point.

This should not happen. We can see that **fst** is not actually used when we go through the **None** arm because a different reference is returned in that case. However, NLL can’t accommodate this situation because **fst** *may* be used later, see Figure 3. A similar problem for map data structures has led to the **Entry** Application Programming Interface (API) which uses unsafe Rust to work around this limitation.

Do the listing again with lifetime annotations?

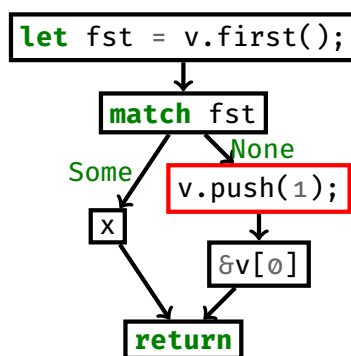


Figure 3: Control-flow diagram for Listing 1.29.

Polonius With lifetimes, we looked *forwards* from the borrow expression to see how long a reference (and therefore the loan) is live. Polonius goes *back-*

wards from a point of use to see if there is still a live reference. Polonius doesn't use lifetimes but *regions* to determine the liveness of a loan.

A *region*⁵ is a set of loans. Each reference has a region consisting of all loans it may depend on. A fresh reference created from an owned value has a region consisting of just one loan, but a reference, e. g. if returned by a function, could depend on several inputs and therefore have a region of several loans. Note that in this step we don't care about how long a reference is valid, we don't go forward in the CFG. Instead, we only consider previous nodes to determine regions.

Now, a loan L is live at some node N , if there is some variable which is live at N and contains L in its region. This difference means that different paths through the CFG are independent from each other, because a node in one path can't see a node in the other one by walking back the CFG.

Let's look at the example from Listing 1.29 with all regions made explicit. $x\{L1, L2\}$ denotes that expression x has a region consisting of the two loans $L1$ and $L2$.

```
fn first_or_insert(v{L0}: &mut Vec<i32>)
-> &{L0, L1, L3} i32
{
  let fst{L0, L1} = v.first(){L0, L1};
  match fst {
    Some(x{L0, L1}) => x,
    None => {v.push(1){L0, L2}; &v[0]{L0, L3}},
  }
}
```

Listing 1.30: Example with region annotations

In Listing 1.30 we can see that there are four relevant loans: $L0$ is the loan of the reference we got passed in. All references depend on it. **first** creates a reference with loan $L1$ that is returned in the **Some** branch, **push** implicitly reborrows to push a value onto $*v$. The final reference is created by the index operation and it may also be returned. Therefore, the return value has a region $\{L0, L1, L2\}$, because those three loans are what it may depend on.

Under NLL, the **push** was not possible because x being live and depending on **fst** meant that **fst** was live. With Polonius, we must check if there is any live variable that has a nonempty intersection with $\{L0, L2\}$. **fst** and x are not live, so they don't pose a problem, even if the regions overlap. v is live and there is a region overlap, but since the compiler inserts a reborrow, it is not a problem. There could still be an error if the borrow stack were invalidated at a later point, but since Polonius is only looking backwards, this is not something we have to consider here. There are no more live variables, so the node passes the borrow check.

⁵Polonius calls regions *origins*, which is a more telling name, but regions is the more standard term.

Self-referential Structs Sometimes we want to have structures that contain references to parts of itself.⁶ Consider the struct in Listing 1.31, in which we store some data and additionally provide a view into a part of the data. The `window` field contains a reference and it must be tied to some value on the stack, but it is impossible to assign a lifetime to it. Putting a `'data` is not valid Rust at the moment.

With regions, however, this could work. Polonius could check that when creating a `View`, the reference supplied to the constructor is pointing to the `data` argument.

```
struct View<T> {
    data: Vec<T>,
    window: &'data [T]
}
```

Listing 1.31: Self-referential struct

Polonius the Crab Since Polonius is still unstable and will be so for the foreseeable future, there is a library⁷ defining a macro which implements the Polonius logic in stable Rust, using some unsafe code.

Cyclone *Cyclone* [Gro+02] is a dialect of C that introduces additional safe pointer types which are checked using regions. Instead of Polonius’ abstract regions, the regions in Cyclone represent concrete parts of memory.

1.2 Contracts and Refinement Types

Every interface, be it a function, a library, or an HTTP server, comes with a *contract*: The interface expects data in a certain, well-defined form, and guarantees—given the input complies with this form—certain properties of the output. On the other hand, if the input is not well-formed, no guarantees on the output are made whatsoever. This principle is called “garbage in, garbage out”. If the interface exposes data, there are normally also some invariants that these data are supposed to conform to for the whole program.

Normally, these contracts are implicit and it is the burden of the author of an interface to carefully check and document them and the burden of the consumer of the interface to carefully follow them. This is not only a lot of work but also a large source for errors. Because of that, many programming languages offer some facilities to help users documenting and enforcing these contracts, the most notable being types. The input and output types of a function are part of its contract that can be extracted by a documentation tool and checked by the compiler.

Strong typing rejects some correct programs but also prevents many potential bugs and thus is used in many languages. The difficulties lie in how a type system should be designed: If a type system is too weak, it can’t find

⁶A real-world example of this are futures which must store the point of execution they are in. Currently they are a special case in the compiler and can’t be expressed in user-code.

⁷https://docs.rs/polonius-the-crab/0.3.0/polonius_the_crab/

mistakes and is not very useful. On the other hand, strong type systems like the calculus of constructions allow programmers to specify contracts down to the last detail and proof that an implementation follows this specification, but are very difficult to use and unwieldy.⁸

citation

In this section, we will focus on two related approaches of specifying contracts, that are very useful without being overly complex: Contract programming inserts explicit runtime checks into a program to ensure that it is never in an illegal state. Refinement types introduce compile-time checks. Finally, we will discuss hybrid typing, an approach to combine contract programming and refinement types.

1.2.1 Contract Programming

When encountering an error or illegal state at runtime, a program can react basically in two ways: Firstly, silently ignoring the error and thus creating undefined behavior. This can lead to wrong results, which can be dangerous if the results are still plausible and so the error is not noticed. It could also lead to memory corruption and thus leak sensitive information.

The second option is to crash the program. This may be annoying but prevents potentially dangerous undefined behavior. While failing a complete system is often not desirable, failing a component is normally preferable over it being corrupted.

Contract programming, or design by contract, is about enforcing program correctness at runtime by crashing if a contract is violated and thus an illegal state would be reached. Some programming languages support contract programming natively and automatically expose contracts as part of a library's interface.

There are mainly three kinds of contract:

1. *Preconditions* are checked at the beginning of some piece of code,
2. *Postconditions* are checked after some piece of code has run, and
3. *Invariants* must always hold for some piece of data. An invariant on a piece of data is equivalent to checking the invariant as both a pre- and postcondition for each computation involving this data.

We can model a bank transfer system as an example: A bank account has a balance with the invariant that it may never below line of credit. One can withdraw money from an account, but only if the account is not overdrawn. When transferring money from one account to another, there is a precondition on the first account to have enough money and a postcondition that the total amount of money hasn't changed. Both the withdrawal and the transfer must use nonnegative **amount**. An implementation in Rust without any contract checking is given in Listing 1.32.

⁸A proof for the correctness of QuickSort in the language Coq spans several hundred lines: <https://gist.github.com/RyanGLScott/ff36cd6f6479b33becca83379a36ce49>

Not really, because the invariant may be violated during the runtime of a method as long as it is not visible outside of the method (which it could be in a parallel shared mutable access situation). In D, invariants are checked after pre- and before postconditions of all public methods.

```

struct Account {
    line_of_credit: i32,
    balance: i32,
}

impl Account {
    fn change_line_of_credit(&mut self, new: i32) {
        self.line_of_credit = new;
    }

    fn withdraw(&mut self, amount: i32) -> i32 {
        self.balance -= amount;
        amount
    }

    fn transfer(
        &mut self,
        &mut other: Self,
        amount: i32,
    ) {
        self.balance -= amount;
        other.balance += amount;
    }
}

```

Listing 1.32: A simple banking system

This version of the code is dangerous because no checks are present whatsoever. A naïve way of enforcing the contracts is to introduce branching: We check that the condition is met, and if it isn't, we don't change an account. But we also need to signal failure by returning a **Result** or **Option**. See Listing 1.33.

```

// Implementation of `std::error::Error` is omitted.
struct BankingError;

struct Account {
    line_of_credit: i32,
    balance: i32,
}

impl Account {
    fn change_line_of_credit(
        &mut self,
        new: i32,
    ) -> Result<(), BankingError> {
        if self.balance >= new {
            self.line_of_credit = new;
            Ok(())
        } else {
            Err(BankingError)
        }
    }

    fn withdraw(
        &mut self,
        amount: i32,
    ) -> Option<i32> {
        if self.balance - amount >= self.line_of_credit {
            self.balance -= amount;
            Some(amount)
        } else {
            None
        }
    }

    fn transfer(
        &mut self,
        &mut other: Self,
        amount: i32,
    ) -> Result<(), BankingError> {
        if self.balance - amount >= self.line_of_credit {
            self.balance -= amount;
            other.balance += amount;
            Ok(())
        } else {
            Err(BankingError)
        }
    }
}

```

Listing 1.33: A simple banking system with naïve checks

This ensures that the invariant for **balance** is not broken and the instead consumer of the API has to manually check if a method call was successful. Also, the possibility of failure is a visible part of the API now. An additional advantage is that this library will not panic if a condition is violated. But it incurs a lot of boilerplate (much of it lies in the implementation of **std::error::Error** for **BankingError** which is omitted here) and repeated code. Also note that there is an additional hidden condition in that the programmer must always keep in mind that **line_of_credit** ≤ 0 .

Reformulate

Assertions While using **Result** and **Option** types is safe and explicit, it comes with a considerable overhead for both the user and the maintainer of a library. Often this overhead is undesirable and the program is allowed to just crash on failure. Assertions can be used for that. **assert!(foo)** checks if **foo** is **true** and panics if not.

```
struct Account {
    line_of_credit: i32,
    balance: i32,
}

impl Account {
    fn change_line_of_credit(&mut self, new: i32) {
        assert!(balance >= new);
        self.line_of_credit = new;
    }

    fn withdraw(&mut self, amount: i32) -> i32 {
        assert!(self.balance - amount >= self.line_of_credit);
        assert!(amount >= 0);
        self.balance -= amount;
        amount
    }

    fn transfer(
        &mut self,
        &mut other: Self,
        amount: i32,
    ) {
        assert!(self.balance - amount >= self.line_of_credit);
        assert!(amount >= 0);
        self.balance -= amount;
        other.balance += amount;
    }
}
```

Listing 1.34: Enforcing a contract using assertions

The assertions in Listing 1.34 again ensure the invariant on **balance**. But now the failure is not part of the API and must be documented separately.

Also, there are still more conditions which are not checked here. Checking those would be very verbose.

Also, it might be desirable to only use assertions during testing but not in production code. For this use case, Rust provides debug assertions that are not compiled in release mode.

Integrated Contract Programming Programming languages like Eiffel and D have native support for contract programming. The `contracts` and `adhesion` crates bring restricted versions of design by contract to Rust.

loop invariants/conditions

Higher-order contracts [Findler, Felleisen](#)

1.2.2 Refinement Types

Since the inception of set theory, mathematicians used *set comprehension* to build sets. For example,

$$\{ n \mid n \text{ is a prime number} \}$$

denotes the set of all prime numbers. On the left-hand side of the $|$ is a comprehension variable and on the right-hand side is a predicate. The above is an example of *unrestricted comprehension* because n can stand for any mathematical object. Allowing unrestricted comprehension leads to Russel's paradox, though, which is why it is not used in modern mathematics anymore. There is a weaker version of comprehension which works well, *restricted comprehension*. In it, the variable has to be restricted to a set, like the following:

$$\{ n \in \mathbb{N} \mid n \text{ is a prime number} \}$$

This means that for every set X and every predicate P ,

$$\{ x \in X \mid P(x) \} \subseteq X$$

which circumvents the paradox. *Refinement types* are the type theoretic analogue to restricted comprehension. Some programming languages allow for refinement types. For example, in the Lean theorem prover, the introductory example would translate to

```
{ n :  $\mathbb{N}$  // prime n }
-- Syntactic sugar for:
subtype nat prime
```

Listing 1.35: Refinement types in Lean

`subtype` is a type constructor, that is a generic type, with two arguments. The first argument `nat` is a type, but the second argument `prime` is a value of function type. Rust has limited support for types depending on values with *const generics*, but it does not support functions as const generics, which means

```

struct Prop<const B: Bool>;

impl<const B: bool> Prop<B> {
    const fn new() -> Self {
        if B {
            Self
        } else {
            panic!()
        }
    }
}

struct Subtype<T, const pred: fn(T) -> bool> {
    val: T,
    proof: Prop<pred(val)>
}

impl<T, const pred: fn(T) -> bool> Subtype<T, pred> {
    const fn new(val: T) -> Self {
        Self {
            val,
            proof: Prop::new::<pred(val)>()
        }
    }
}

// Also implement `Deref` and `DerefMut` so that a
// `Subtype<T, pred>` can be used in any place where a
// `T` is expected.

```

Listing 1.36: The Rusty equivalent.

that refinement types can't be implemented in Rust this way. There have been proposals for introducing more const generic capabilities to Rust⁹, but they are put into cold storage indefinitely. We can still imagine what the equivalent Rust code would be if it could be written:

Prop is a helper type which can only be created from a true value. We need this so that we can proof that **val** makes **prop** true. This would now allow us to create constants that have certain proven properties like in Listing 1.37.

```

const PRIME: Subtype<i32, is_prime> = Subtype::new(17);
const SMALL: Subtype<u32, |x| x <= 255> = Subtype::new(9);

```

Listing 1.37: Constants with proven properties

Because this is cumbersome, it would also be nice to extend the syntax for

⁹<https://github.com/ticki/rfcs/blob/pi-types/text/0000-pi-types.md>

More on full dependent types, Pi and Sigma types, lambda cube?

easier definition of a refinement type and let the compiler automatically insert the `Subtype::new`.

```
const PRIME: i32 where is_prime = 17;
const SMALL: u32 where |x| x <= 255 = 9;
```

Listing 1.38: Constants with proven properties

Refinements could of course only be checked at compiletime for constants. If a user wants to create a runtime value, the checks are also at runtime.

Refinement Types in Functions In the previous example, the predicate only took one input, namely the comprehension variable itself. And as long as only simple types are refined, this is the only possibility. But refinement types really shine when they depend on multiple inputs. The prime example is a function concatenating two arrays:

```
fn concat<T, const N: usize, const M: usize>(
    first: &[T; N],
    second: &[T; M],
) -> [T; N + M] {
    // ...
}
```

Listing 1.39: Concatenate two arrays, with a dependent output type

Here, the type of the return value depends on the generic values of the inputs. This, again, only works if `N` and `M` are known at compile time.

Runtime dependent function (e. g. `reverse?`)

Intersection types

1.2.3 Liquid Types and SMT Solving

1. Refinement Types are not decidable
2. SAT is decidable
3. SMT is decidable
4. Liquid Types = $\{T : \text{RefinementType} \mid \text{decidable } T\}$
5. SMT-LIB2 and Z3

1.2.4 Hybrid Contract Checking

1. Idea: Combine Contracts and Liquid Types
2. Sprinkle in casts everywhere an assertion is made

3. If the compiler can prove that this is ok or not ok, remove the cast (and potentially throw a compiler error)
4. All other casts are dynamic and lead to a panic if they fail.
5. In Rust, `Result` and `?` could be used instead. Problem: The error must be handled somewhere.

1.3 Other Extensions to a Type System

1.3.1 Totality and Termination Metrics

1. Some functions can panic or loop indefinitely
2. In Rust, `!` as return type helps (such a function does definitely not terminate)
3. There is no “may not terminate”, though, so you have to rely on the documentation
4. You can prove that a loop/recursion finishes by termination metrics
5. Panics can technically occur in every function (every function call needs memory space for a stack frame)
6. But you can prove that there are no “high-level” panics in any execution path within a function

1.3.2 (Other) Substructural Types

1. What are structural rules? Exchange, Weakening, Contraction
2. Move semantics are affine types
3. Clone and Copy is weakening
4. `#[must_use]` and relevant types
5. Linear types
6. The borrow stack is ordered types
7. Uniqueness types: There is only one reference (in contrast to linear types: no more references can be created)

Proof objects? Compile-time only linear values

1.3.3 Purity and (algebraic) Effects

1. Effects
2. Handling (Example: Exceptions)
3. Algebraic Effects
4. Pure functions don't have unhandled effects

5. What counts as an effect (IO, allocation, computation)
6. non-totality is an effect, but do termination metrics fit in?
7. Weakening and Contraction are effects, but what about borrowing?
8. Refinements \cap Effects

Proof objects, proof search, provable pointers, existential types, proof for non-aliasing à la ATS and an improved borrow checker?

Intermediate Representations and MIR

Flow-sensitive typing; for example:

extract: $\lambda\{x: \text{Option}\langle T \rangle \mid x.\text{is_some}()\}$ $\rightarrow T$ or the ? operator using extract if possible

```
// Note that the return type is `()`
fn do_stuff_if_some<T : Debug>(x : Option<T>) {
    if x.is_none() {
        return;
    }
    println!("{:?}", x?); // Infallible at this point
}
```

2 Formal Semantics for Combining OBS and Refinement Types

Our goal is to combine Ownership and Borrow System and refinement types introduced in chapter 1 into one semantics. In this chapter, we survey different kinds of formal semantics in general and semantics that already exist for Rust. We then discuss their amenability to extension with refinement types.

2.1 Kinds of Formal Semantics

1. Operational Semantics
 - a) big-step SOS
 - b) MSOS
 - c) small-step SOS
 - d) Reduction Semantics
 - e) Abstract Machine Semantics
2. Denotational Semantics
 - a) Direct Style
 - b) Continuation-Passing Style
 - c) Continuation-Passing and RustBelt
 - d) Monadic Semantics
3. Axiomatic Semantics and Hoare Logic
4. Action Semantics

2.2 Survey of Formal Semantics for Rust

1. RustSem
 - a) language-independent Operational Semantics for OBS
 - b) Works on the *Core Language* IR, which is closer to Rust than MIR
 - c) Implemented in the K framework
2. Oxide
 - a) Not concerned with **unsafe**, concurrency, sum types or traits
 - b) Region-based
 - c) Works on *Oxide*, with an operational semantics on top of it
3. Patina
 - a) Syntactic Borrow Checker
 - b) Works on a pre-1.0 version of Rust
4. Featherweight Rust
 - a) *FR* only does borrow checking, not typing
 - b) No NLL, closures, Branching, Products, or **unsafe**
 - c) Small-step Operational Semantics
5. KRust
 - a) Uses the K Framework
 - b) No traits/pattern matching
 - c) Term rewriting system
6. RustHorn
 - a) Uses Constrained Horn Clauses
 - b) Problems with pointers
7. Polonius
 - a) Operates on MIR/the CFG
 - b) Constraint solving using Datafrog
8. Ferrocene
 - a) Limited formal specification
 - b) Only really a syntax for now, not a semantics

2.3 Discussion

1. Most semantics are too complicated
2. Polonius follows the control flow directly
3. Refinement type checking is also control flow sensitive
4. There is no formal semantics of Polonius, only a Datafrog model
5. Therefore, we need to transform Polonius into a “real” formal semantics
6. Discussion of which Semantics to use

Abbreviations

RAII	Resource Acquisition is Initialization	2
API	Application Programming Interface	18
SBRM	Space Bound Resource Management	4
NLL	Non-Lexical Lifetimes	1
CFG	control-flow graph	12
MIR	Mid-level Intermediate Representation	15
RFC	Request For Comments	
TPB	Two-Phase Borrow	1
GC	garbage collection	2
OBS	Ownership and Borrow System	2
HTTP	Hypertext Transfer Protocol	

List of Listings

1.1	Memory management in C	3
1.2	Manual resource management in Python	3
1.3	Context managers in Python	3
1.4	The defer statement in Go	4
1.5	RAII in C++	4
1.6	Heap allocation in Rust	4
1.7	Ownership transfer	5
1.8	Usage of Clone	6
1.9	Number types are Copy	6
1.10	Borrow using references to prevent a move	7
1.11	Lexical Lifetimes prevent usage	8
1.12	More scopes make the code check	9
1.13	Try to return a reference	10
1.14	Borrow-through	10
1.15	Iterator over two slices	10
1.16	Iterator over two slices with lifetime parameters	11
1.17	The output lifetime depends only on one of the inputs	11

1.18	Use String and as_ref in place of [u8]	11
1.19	Use String in place of [u8]	12
1.20	Simple NLL example	12
1.21	NLL example with bracing	13
1.22	Cannot create a shared reference while an exclusive one exists .	14
1.23	Move a borrow into a function	14
1.24	Reborrow through an exclusive reference	15
1.25	Borrow twice in one method call	15
1.26	Workaround for Listing 1.25	16
1.27	Example of inline TPB	16
1.28	Indexing twice into a vector is illegal	17
1.29	NLL reject correct program	18
1.30	Example with region annotations	19
1.31	Self-referential struct	20
1.32	A simple banking system	22
1.33	A simple banking system with naïve checks	23
1.34	Enforcing a contract using assertions	24
1.35	Refinement types in Lean	25
1.36	The Rusty equivalent.	26
1.37	Constants with proven properties	26
1.38	Constants with proven properties	27
1.39	Concatenate two arrays, with a dependent output type	27