YOU CAN'T SPELL TRUST WITHOUT RUST

ALEXIS BEINGESSNER

A thesis submitted to the Faculty of Graduate and Post Doctoral Affairs
in partial fulfillment of the requirements for the degree of

Master's in Computer Science

Carleton University
Ottawa, Ontario, Canada

ABSTRACT

Rust is a new programming language developed by Mozilla in response to the fact that C and C++ are unsafe, inefficient, and unergonomic –particularly when applied to concurrency. Version 1.0 of Rust was released in May 2015, and appears to be performing excellently. Rust code is memory-safe by default, faster than C++, easier to maintain, and excels at concurrency.

Yet little analysis exists of the semantics and expressiveness of Rust's type system. This thesis focuses on one of the core aspects of Rust's type system: *ownership*. Ownership is a system for expressing where and when data lives, and where and when data can be mutated. In order to understand ownership and the problems it solves, we provide a novel analysis of both in terms of *trust*.

We observe that many classical memory safety errors are the result of some code trusting data to be in a particular state that doesn't necessarily hold. For instance, indexing out of bounds occurs when one incorrectly trusts an index to agree with an array's length. The problem of trust is largely addressed using three different strategies: *naivety*, *paranoia*, and *suspicion*. Each of these strategies represents a compromise among the competing concerns of control, safety, and ergonomics. Briefly, naive interfaces optimize for control by assuming that state is correct; paranoid interfaces optimize for safety by statically preventing invalid states; and suspicious interfaces optimize for ergonomics by validating states at runtime.

We demonstrate how ownership can be used to build efficient and ergonomic paranoid interfaces, even when building on top of unsafe naive interfaces. For example, iterators in Rust provide an interface for indexing into an array without bounds checks while being statically immune to invalidation. Two of these interfaces, *drain* and *entry*, were developed by us for Rust's standard library.

Finally we demonstrate some of the limits of ownership. In particular, we explore how the lack of proper linear typing makes it difficult to guarantee that desirable operations are performed, such as freeing unused memory or repairing invariants. However we provide some tricks for *ensuring* that operations are performed (at the cost of ergonomics), or simply mitigating the impact of not performing them.

CONTENTS

INTRODUCTION

Modern systems are built upon shaky foundations. Core pieces of computing infrastructure like kernels, system utilities, web browsers, and game engines are almost invariably written in C and C++. Although these languages provide their users the control necessary to satisfy their correctness and performance requirements, this control comes at the cost of an overwhelming burden for the user: *Undefined Behaviour*. Users of these languages must be ever-vigilant of Undefined Behaviour, and even well-maintained projects fail to do so [25]. The price of invoking Undefined Behaviour is dire: compilers are allowed to do *absolutely anything at all*.

Modern compilers can and will analyze programs on the assumption that Undefined Behaviour is impossible, leading to counter-intuitive results like time travel, wherein correct code is removed because subsequent code is Undefined. This aggressive misoptimization can turn relatively innocuous bugs like integer overflow in a print statement into wildly incorrect behaviour. Severe vulnerabilities subsequently occur because Undefined Behaviour often leads to the program blindly stomping through memory and bypassing checks, providing attackers great flexibility.

As a result, our core infrastructure is constantly broken and exploited. Worse, C and C++ are holding back systems from being as efficient as they could be. These languages were fundamentally designed for single-threaded programming and reasoning, having only had concurrency grafted onto their semantics in the last 5 years. As such, legacy code bases struggle to take advantage of hardware parallelism. Even in a single-threaded context, the danger of Undefined Behaviour encourages programs to be written inefficiently to defend against common errors. For instance it's much safer to copy a buffer than have several disjoint locations share pointers to it, because unmanaged pointers in C and C++ are wildly unsafe.

In order to address these problems, Mozilla developed the Rust programming language. Rust has ambitious goals. It intends to be more efficient than C++ and safer than Java without sacrificing ergonomics. Although Rust 1.0 was released less than a year ago [9], early results are incredibly promising. To see this, we need look no further than the Servo project, a rewrite of Firefox's core engine in Rust [11].

Preliminary results [14] have found that Servo can perform layout two times faster than Firefox on a single thread, and four times faster with four threads. Servo has also attracted over 300 contributors [12], in spite of the fact that it's not production ready and written in a language that was barely just stabilized. This demonstrates that Rust has *some* amount of usability, and that applications written in it have can actually be maintained by novices (since almost everyone using Rust is new to it). There's little data on safety and correctness in Servo, but we'll see throughout this thesis that Rust provides powerful tools for ensuring safety and correctness.

It should be noted that Rust doesn't desire to invent novel systems or analyses. Research is hard, slow, and risky, and the academic community has already produced a wealth of rigorously tested ideas that have little main-stream exposure. As a result, to a well-versed programming language theorist Rust can be easily summarized as a list of thoroughly studied ideas. However, Rust's type system as a whole is greater than the sum of the parts. As such, we believe it merits research of its own.

Truly understanding Rust's type system in a rigorous way is a massive undertaking that will fill several PhD theses, and Derek Dreyer's lab is already working hard on that problem [10]. As such, we will make no effort to do this. Instead, we will focus in on what we consider the most interesting aspect of Rust: *ownership*. Ownership is an emergent system built on three major pieces: *affine types*, *region analysis*, and *privacy*. Together we get a system for programmers to manage where and when data lives, and where and when it can be mutated, while still being reasonably ergonomic and intuitive (although with an initially steep learning curve).

Ownership particularly shines when one wants to develop concurrent programs, an infamously difficult problem. Using Rust's tools for generic programming, it's possible to build totally safe concurrent algorithms and data structures which work with arbitrary data without relying on any particular paradigm like message-passing or persistence. Rust's crowning jewel in this area is the ability to have child threads safely mutate data on a parent thread's stack without any unnecessary synchronization, as the following program demonstrates. (If you aren't familiar with Rust, a brief introduction is provided in the Appendix)

```
[dependencies]
crossbeam = 0.1.6

-----

extern crate crossbeam;

fn main() {
    // An array of integers, stored on the stack
    let mut array = [1, 2, 3];

    // Create a scope to know when to join all threads
    crossbeam::scope(|scope| {

        // Get pointers to each element in the array
        for x in &mut array {
            // Spawn a thread to increment this element
            scope.spawn(move || {
                *x += 1;
            });
        }

        // Block on all the child threads joining here
    });

    println!("{:?}", array);
}
```

The most amazing part of this program is that it's based entirely on a third- party library (crossbeam [3]). Rust does not require threads to be modeled as part of the language or standard library in order for these powerful safe abstractions to be constructed. All of the following operations won't just fail to work, but will *fail to compile*: sharing non-threadsafe data with the scoped threads, giving the same pointer to two threads, keeping a pointer for too long, or accessing the array while the threads are still running. The fact that misuse is a static error is incredibly important for concurrent programs, because runtime errors can be incredibly difficult to reproduce or debug.

In order to understand how this program works and why it's safe, we'll need to build up some foundations.

In chapter 2, we establish the basic principles of safety and correctness that we're interested in. Since these are ultimately vague concepts in a practical setting, we do this by surveying several classic errors that all languages must deal with in one way or another.

In chapter 3 we provide a novel analysis of these problems in terms of *trust*. In particular, we observe that these problems can be reduced to how code trusts other code with data. We identify three major strategies for handling these trust issues, and briefly categorize some common solutions to these problems according to these strategies.

In chapter 4 we introduce ownership. We show how affine types, region analysis, and privacy contribute to Rust's ability to model and enforce trust between different pieces of code. Although this is mostly describing concepts that are well-understood at an intuitive level in the Rust community, this work is the first to actually write them down completely, and in terms of trust.

In chapter 5 we analyze several standard Rust interfaces use ownership for trust. We demonstrate how Rust's story surrounding indexing into arrays exactly aligns with our strategies for trust. Of interest is that tagged unions use ownership to provide values that must be checked for validity to be accessed, and iterators use ownership to provide safe unchecked indexing. We use the *entry* API, which was developed as part of this work, to demonstrate how ownership can be used to externally expose the execution of an algorithm, producing more flexible designs that gives greater control to users.

In chapter 6 we discuss some of the limitations of Rust's version of ownership. We show how ownership struggles with ad-hoc cyclic data structures, and some of the community's solutions to this problem. We use the *drain* API, another interface developed as part of this work, to demonstrate how ownership is mostly only able to model things that *must not* be done by an interfaces user, but struggles to model things that *must* be done. Finally, we review how the scoped thread API works.

In chapter 7 we briefly survey how ownership in Rust compares to similar systems in other languages. In particular, Cyclone and C++, Rust's two closest relatives.

2 SAFETY AND CORRECTNESS

Programming is hard. [citation needed]

We would like our programs to be correct, but we seem to be very bad at doing this. In fact, any assertion that a non-trivial program is totally correct will be met with deep suspicion. This isn't unreasonable: any non-trivial property one might be interested in is usually *undecidable*, even in trivial programming languages. Still, some bugs are more pervasive and pernicious than others. It's one thing to get a poorly rounded output, but another thing altogether to join a botnet because of a sloppy integer parser.

For this reason, it is common to distinguish one major category of bug from all others: *memory safety*. A system is said to be memory-safe if access to memory is properly restricted. This is an unfortunately vague definition, but that's because it's a broad problem. At the heart of memory safety are two major concerns: preventing secrets from being stolen, and systems from being hijacked. It is of course possible to design a memory-safe system that liberally leaks secrets and happily takes arbitrary commands from the public internet, but an absence of memory safety makes even innocuous systems trivial to exploit in this way.

Most importantly, programs must have *some* restriction on what can be read and written by what. If a function can freely read and write all the memory in a process, then compromising that function can almost certainly make it do anything and everything. In particular, overwriting return pointers in a program's call stack can often enable an attacker to execute arbitrary computations, even in a system with proper code-data separation and address space layout randomization. [17]

Even "only" being able to read arbitrary memory can be an exploit, particularly as reading memory often implies writing the value somewhere else. This is the basis for the infamous Heartbleed attack, where clients could ask servers to send them random chunks of its memory, leaking the server's private key [22].

Upholding memory safety is a difficult task that must often be approached in a holistic manner, because two interfaces that are memory-safe on their own can easily be unsafe when combined. For instance, dereferencing a pointer that is known to be valid is safe, and offsetting a pointer is technically safe (it's just adding integers), but being allowed to dereference *and* offset is unsafe, because this allows arbitrary memory to be accessed.

In addition to the matter of memory safety, there is also the broader problem of overall correctness. It is of course impossible for a system to enforce that all programs are correct, because it requires the programmer to correctly interpret and encode their program's requirements. Computers can't possibly understand if requirements reflect reality, so they must trust the programmer in this regard. That said, it is possible for a system to understand *common* errors. Where it's impractical to prevent these errors in general, a language can orient itself to make these errors less likely. For instance, few programs expect integers to overflow, so a system that does more to prevent integer overflow or mitigate its consequences may be regarded as safer.

Rust is fundamentally committed to being memory-safe. If memory safety can be violated, this is a critical issue in Rust's design and must be fixed. That said, Rust also tries to be safer in general. All else equal, if an interface can prevent misuse, it should. The question is if all else is *really* equal, or if preventing an error comes at significant ergonomic or performance costs.

While the techniques demonstrated in this thesis are can be applied generally, we will be focusing on a few particular common problems that can lead to memory safety violations: use-after-free, indexing out of bounds, iterator invalidation, and data races. In addition, we will consider the problem of memory leaks, which aren't a memory-safety problem, but still problematic.

We will see how all of these problems occur in the two most prolific programming languages of all: C and C++. It's worth noting that the practices encouraged by "modern" C++ can significantly mitigate or eliminate these errors, but unfortunately these are only conventions, and must be rigorously followed and understood to work correctly. A C++ programmer who is unaware of these practices gains little safety from using the latest C++ release, particularly since the unsafe practices are often the most convenient and well-supported.

2.1 USE-AFTER-FREE

A use-after-free is one of the two canonical memory safety violations. It occurs when some memory is marked as unused (freed), but the program continues to use that memory. Here's two simple versions of this problem, as seen in C:

```c
// use-after-free with heap allocation

#include <stdlib.h>
#include <stdio.h>

int main() {
    // Get some memory from the allocator
    char* string = (char*) calloc(3, 1);
    string[0] = 'h';
    string[1] = 'i';

    // Give the memory back to the system allocator
    free(string);

    // Use the memory anyway
    printf("%s\n", string);
}

// use-after-free with stack allocation

#include <stdio.h>

int main() {
    char* ptr = 0;

    if (!ptr) {
        char data = 17;
        // take a pointer to `data`
        ptr = &data;
    }
    // data is now out of scope, and undefined

    // read it anyway
    printf("%u\n", *ptr);
}
```

In principle, a use-after-free can be harmless if the code in question gets lucky. The system may never look at this memory again, allowing it to continue to behave in an allocated manner. However the system has been given permission to do whatever it pleases with this memory, which can lead to several problems.

First, the memory can be returned to the operating system. If such memory is used, this will cause a page fault, causing the program to crash. This is annoying, but also arguably memory-safe in the sense that the memory is never successfully used after it is freed. The problematic case is when the allocator *reuses* the memory. Any subsequent allocation may receive the freed memory, leading the memory to be used for two different purposes. This violates memory safety, allowing code to read or write memory that it wasn't supposed to.

Use-after-frees are solved by almost all languages by requiring pervasive garbage collection. C and C++ are the two major exceptions, which by default don't use any garbage collection mechanisms. In these two languages use-after-free is a rich source of exploits to this day. As we will see, this is no coincidence; the solution taken by most languages is unacceptable to those who traditionally use C or C++.

Use-after-frees are the canonical example of the more general problem of using something after it has been logically destroyed. For instance, a similar problem applies to using file handles after they're closed. However the use-after-free is particularly problematic because most system allocators pool and reuse pages of memory acquired from the operating system, which prevents it from understanding misuse.

File handles, on the other hand, are generally not pooled like this. That said, it's possible for new handles to match the value of old handles (like pointers, they're ultimately just integers). This can lead to a similar scenario as a use-after-free, where two distinct handles are independently trying to use the same file.

The more general category reveals a fundamental weakness of garbage collection: it only solves a use-after-free, and no other kind of use-after. Although all resources could be garbage collected, it's common to manually manage many other resources because they're much more limited, and have externally observable effects. For instance, file writes may not be committed to disk until the file is properly closed. Unfortunately, allowing manual management necessitates misuse to be guarded against at runtime. Since any program that tries to use-after is certainly *wrong*, it's desirable to prevent programs that do this from compiling at all.

2.2 INDEX OUT OF BOUNDS

An index out of bounds is the second canonical memory safety violation. It occurs when the index used to access an array is too large or small. A simple C version:

```c
#include <stdio.h>

int main() {
    // An array of 5 elements
    char data[5] = {1, 2, 3, 4, 5};

    // Print out the 17th element
    printf("%u\n", data[17]);
}
```

The consequence of an index out of bounds is quite straight-forward: it reads or writes memory that happens to be near the array. For arrays sitting on the stack, this is an easy vector for overwriting a function's return pointer.

Like general use-afters, this problem is usually resolved through runtime checks. Simply have every operation that takes an index compare it to the length of the array. However C and C++ provide no guards by default, and in some cases array length isn't even known, making it *impossible* to check.

2.3 ITERATOR INVALIDATION

Iterator invalidation occurs when a collection is mutated while it's being iterated over. It's a particularly interesting problem because it's similar to a use-after-free or indexing out of bounds, but can require more pervasive checking to guard against. A simple example in C++:

```cpp
#include <vector>
#include <stdio.h>
using namespace std;

int main() {
    // a growable array of four 5's
    vector<char> data (4, 5);

    // loop over the array to print the elements out
    for (auto it = data.begin(); it != data.end(); ++it) {
        printf("%u\n", *it);

        // but push a single element onto
        // the stack while doing it
        if (it == data.begin()) {
            data.push_back(0);
        }
    }
}
```

When an iterator is invalidated it will start acting on outdated information, potentially stomping through memory that is no longer part of the collection. For instance, when we executed this program it produced 22(!) values. Evidently the array's backing storage had been reallocated in a different location on the heap, conveniently only a few bytes over. The *it* variable simply walked forward through the heap until it found the new end of the array. In other words, we managed to index out of bounds *and* use-after-free at the same time.

Even if an iterator isn't invalidated, mutating a collection while iterating it is an easy way to produce a nonsensical program. For instance, unconditionally inserting into a collection while iterating into it may lead to an infinite memory-consuming loop. That said, it can be be desirable to mutate a collection while iterating it. Removing elements while processing them is a rather common operation.

In order to guard against this, any operations which can invalidate an iterator need to somehow know about any outstanding iterators, or otherwise signal to them that a change has been made. One way to do this is to have the collection contain a timestamp of when it was last modified, and have iterators repeatedly verify that it hasn't changed. This is exactly how OpenJDK's ArrayList is implemented [5].

Once again, neither C nor C++ provide any guards against this sort of thing by default. Even in Java there's no automatic solution to this problem. The programmer just needs to think about iterator invalidation and ensure that it can't happen.

Iterator invalidation can be generalized to the invalidation of any "view" into another type. Iterators are particularly nasty because they're often low-level and performance sensitive, but any view can be made inconsistent by mutating what it intends to represent.

2.4 DATA RACES

A data race occurs when two separate threads attempt to access the same location in memory in an unsynchronized manner, and one of them is writing.

The consequences of a data race are very dependent on details of the implementation and execution, but the values read and written will almost certainly be inconsistent. This can be because of the underlying hardware, or the compiler itself.

The hardware may produce incorrect results because it's trying its hardest to avoid synchronizing the caches and buffers of each individual core. It's quite easy to see this effect by trying to have multiple threads increment an integer repeatedly. One will likely find that several of the increments simply *didn't happen*, producing a smaller result than expected. To guarantee that the increments happen, they must be performed with special atomic instructions.

The compiler may produce incorrect results because it's trying to reason about observable effects and optimize instruction dispatch. It may reason that re-ordering two instructions makes better use of the hardware, but if those two instructions correspond to "lock the mutex" (atomic) and "access the locked data" (non-atomic), the reordering will lead to a data race on the locked data.

Correctly preventing data races is an incredibly difficult problem. As always, C and C++ generally leave this up to the programmer. They expose intrinsics for performing atomic operations that inhibit optimizations and emit atomic hardware instructions, but nothing prevents misusing these operations or simply ignoring them completely. Some systems solve this problem by simply forbidding one of the ingredients of a data race: sharing (message passing), mutability (pure functional programming), or concurrency (JavaScript-style event loops). Java is notable for allowing all three, but still eliminating data races by making aggressive guarantees about atomicity [15].

However Java programs still permit the more general problem of *race conditions*, which are much more difficult to address. A race condition is any situation in which the order in which two threads run their instructions leads to incorrect behaviour. Even if all instructions are atomic, it's quite easy to have a race condition. For instance, if two threads are trying to insert into a growable array, they may both atomically read the length of the array, write to the end of the array, and then update its length. However this valid execution will produce incorrect results:

```
thread A: read length
thread B: read length
thread A: write to end
thread A: increment length
thread B: write to end
thread B: increment length
```

In this execution thread B will overwrite the element inserted by A, and the length will be incremented twice, leading to uninitialized memory being exposed in the last index of the array. This can easily lead to a memory-safety violation if the uninitialized element is read, even though no data races occurred.

2.5 MEMORY LEAKS

Memory leaks are unlike any of the other errors above. The other errors are a matter of doing a thing that shouldn't have been done. Leaks, on the other hand, are failing to do something that *should* have been done. In contrast to a use-after-free, a memory leak occurs when memory is *never* returned to the allocator. We can demonstrate a memory leak by just removing the call to 'free' from our use-after-free example:

```c
#include <stdlib.h>
#include <stdio.h>

// memory leak with heap allocation
int main() {
    // Get some memory from the allocator
    char* string = (char*) calloc(3, 1);
    string[0] = 'h';
    string[1] = 'i';

    printf("%s\n", string);

    // Don't bother to release the memory
}
```

Leaks aren't a memory-safety issue. Leaks just waste resources which can seriously impact performance, or even cause a crash. Crashing is obviously not good, so leaks should be avoided if possible. That said, leaks can be benign or even desirable under the right circumstances. For instance, our example of a leak is actually totally fine. Operating systems reclaim a process' memory on exit anyway, so fiddling with the allocator is just slowing down the program's execution for no good reason.

Most languages handle memory leaks with garbage collection. C++ primarily handles memory leaks with destructors. C doesn't provide anything for avoiding memory leaks. Regardless, no strategy is perfect because properly eliminating leaks may require semantic understanding of how an application works. For instance, none of the strategies mentioned can deal with forgetting to remove unused values from a collection. Even ignoring these "undetectable" cases, these strategies can also fail to properly collect truly unreachable memory due to implementation details.

Memory leaks generalize to the leaking of basically any other resource such as threads, file descriptors, or connections. Arguably, leaks are just a special case of forgetting to do any final step at all. Garbage collection does nothing for these errors, while destructors work just as well for them all.

Leaks are a sufficiently special case that we'll be skipping over them for the bulk of this thesis. For now we'll only focus on the errors that can lead to programs behaving incorrectly, instead of just poorly.

3 TRUST

We believe that the memory-safety errors we described in the previous section, and many more like them, all boil down to a single issue: data trust. All programs require or expect data to have certain properties. This trust is so pervasive that it's easy to forget that we're even making assumptions about how data will behave. This trust has many aspects:

First, there is the *property* of the data that is trusted. The simplest form of trust is trusting that certain values don't occur: booleans are either 'true' or 'false', but never 'FileNotFound'. This kind of property is generally quite manageable because it's easy to verify on demand.

The more complex property is trusting that separate pieces of data agree. This is the kind of property that we see in our memory-safety errors. Indices need to agree with the length of the array, the allocator needs to agree that dereferenced pointers are allocated, and iterators need to agree with the collection they iterate. CPUs need to agree on the value stored in memory. This is the kind of trust we will be focusing on for the bulk of this work.

Second, there is the *who* to trust with data. For most applications, it's reasonable to design an interface that assumes that the language, hardware, and operating system are well-behaved. Of course, bugs in any of these systems may necessitate workarounds, but we don't consider this aspect to be particularly interesting. At some limit, these systems just work the way they work, and any oddities can be regarded as a nasty corner case to be documented.

The aspect of who to trust that we consider really interesting is within the boundary of a process. Within a single program, functions need to decide whether to trust other functions. It's fairly reasonable to trust a single concrete function to be well-behaved. For instance, if one wishes to invoke a print function that claims to only read its inputs, it's reasonable to rely on that fact. We consider it reasonable to extend our trust here because it's being done in a *closed* manner. That is, it's in principle possible to test or otherwise verify that the print function doesn't mutate its input, especially if the source is available.

The problematic case is when we need to extend our trust in an *open* manner. Public functions can be called by arbitrary code, and as such need to worry about trusting their caller. Do we trust the caller to pass us correct inputs? Do we trust the caller to invoke us at the right time? Similarly, generic or higher-order functions need to worry about trusting the arbitrary functions that they're given. If we were *given* a printing function instead of getting to call some concrete one, we then have to worry if that print function will uphold its promise not to mutate the input.

Third, there is the *consequences* of the trust; what happens if trust is violated? Consequences for an application can range from marginal to catastrophic; it could perform slightly worse, produce nonsensical output, crash, delete the production database, or compromise the system it's running on. The consequences of an assumption can easily shape whether that assumption is made at all. Certainly, leaving the system vulnerable is not something to be taken lightly.

Fourth and finally, there is the *justification* of the trust. At one extreme, one may simply blindly trust the data. Indexing is unchecked; pointers are assumed to be allocated; and booleans are assumed to be true or false.

At the other extreme, one can statically guarantee that data is correct. Static type checking can help ensure that a boolean doesn't contain FileNotFound, control flow analysis can determine if a pointer is used after it's freed, and normalized databases ensure that there is a single source of truth for each fact.

Somewhere in between these two extremes, one can *validate* or *correct* data at runtime. Indices may be checked against an array's length; inputs may be escaped; and iterators can check that their collection hasn't been modified.

We respectively call these three approaches *naivety*, *paranoia*, and *suspicion*. As we will see, they each have their own strengths and weaknesses that makes them appropriate for different tasks. As such, basically every piece of software relies on a combination of these strategies to strike a balance between the competing demands of control, safety, and ergonomics. That said, some systems clearly favour some strategies more than others.

3.1 NAIVETY

A naive interface isn't a fundamentally bad interface. It's just one that trusts that the world is great and everyone is well-behaved. Providing a naive interface is a bit like giving everyone you meet a loaded gun and trusting that they use it wisely. If everyone *is* well-behaved, then everything goes great. Otherwise, someone might lose a leg.

First and foremost, a naive interface is simply the easiest to *implement*. The implementor doesn't need to concern themselves with corner cases or invalid input, because they are simply assumed to not occur. This in turn gives significant control to the user of the interface because they don't need to worry about the interface checking or mangling inputs. It's up to the user to figure out how correct usage is ensured. Since the user has the most context, they're the best poised to identify when assumptions can or can't be made.

Naive interfaces may also expose lower-level details more freely, on the assumption that they will be used correctly. This empowers users to efficiently and reliably perform operations that may never have been envisioned by the original implementor, or were simply deemed too much work to develop and maintain. Control is absolutely the greatest strength of naive interfaces. They empower excellent programmers to produce excellent code where it matters.

For similar reasons, naive interfaces are also, in principle, ergonomic to work with. In particular one is allowed to try to use the interface in whatever manner they please. This enables programs to be transformed in a more continuous manner. Intermediate designs may be incorrect in various ways, but if one is only interested in exploring a certain aspect of the program that doesn't depend on the incorrect parts, they are free to do so.

The uncontested champion of this approach is C. In particular, its notion of Undefined Behaviour is exactly naivety. C programs are expected to uphold several non-trivial properties, and the C standard simply declares that if those properties *don't* hold, the program is *undefined*. This naivety can be found in relatively small and local details like unchecked indexing, as well as massive and pervasive ones like manual memory management.

Undefined Behaviour demonstrates the extreme drawback of naive interfaces. Breaking C's trust has dire consequences, as the compiler is free to misoptimize the program in arbitrary ways, leading to memory corruption bugs and high severity vulnerabilities. Modern compilers try to give a helping hand with debug assertions and static analysis, but blind trust is evidently too deeply ingrained into C. Programs written in languages like C continue to be exploited, and no end to this is in sight.

If the consequences of misuse are high or the contracts that must be upheld are too onerous, the ergonomic benefits of naive interfaces can be undermined. Having to be always vigilant against incorrect usage can be mentally exhausting. The problems that result from misusing a naive interface can also be cryptic and difficult to debug, due to the fact that they tend to silently corrupt the system, rather than causing an immediate crash.

That said, the control that C's trusting design provides is highly coveted. Operating systems, web browsers, game engines, and other widely used pieces of software continue to be primarily developed in C(++) because of the perceived benefits of this control. Performance is perhaps the most well-understood benefit of control, but it can also be necessary for correctness in e.g. hardware interfaces. Languages without this control are simply irrelevant for many projects.

3.2 PARANOIA

Paranoid designs represent a distrust so deep that one believes things must be taken away so as to eliminate a problem *a priori*. Where the naive would give everyone a loaded gun, the paranoid are so concerned with people getting shot that they would try to make it impossible by eliminating bullets, guns, or even people.

There are two major families of paranoid practices: omissions and static analysis.

Omissions represent pieces of functionality that could have been provided or used, but were deemed too error-prone. The much-maligned *goto* is perhaps the most obvious instance of a paranoid omission, but it's not the most interesting. The most interesting omission is the ability to manually free memory, which is one that all garbage collected languages effectively make. Some less popular examples include always-nullable pointers (Haskell, Swift, Rust), concurrency (JavaScript), and side-effects (pure functional programming).

Static analysis consists of verifying a program has certain properties at compile-time. Static typing is the most well-accepted of these practices, requiring every piece of data in the program to have a statically known type. Various static lints for dubious behaviours also fall into this category, including code style checks. This analysis also includes more exotic systems like dependent-typing, which expresses requirements on runtime values (such as $x < y$) at the type level.

The primary benefit of paranoid practices is maximum safety. Ideally, a paranoid solution doesn't just handle a problem, it *eliminates* that problem. For instance, taking away memory-freeing completely eliminates the use-after-free, because there is *no* "after free" (garbage collection being "only" an optimization). Similarly, a lack of concurrency eliminates data races, because it's impossible to race. Lints may completely eliminate specific mistakes, or simply reduce their probability by catching the common or obvious instances. Dependent typing can eliminate indexing out of bounds, by statically proving that all indices are in bounds [26].

The cost of this safety is usually control or ergonomics. Paranoid practices point to the pervasive problems that unchecked control leads to, and declare that *this is why we can't have nice things*. Certain operations must be forbidden or require burdensome annotation. It's worth noting that a loss of control does not necessarily imply a loss in performance. In fact, quite the opposite can be true: an optimizing compiler can use the fact that certain things are impossible to produce more efficient programs [18].

That said, paranoid practices *do* generally take away a programmer's ability to directly construct an optimized implementation. One must instead carefully craft their program so that it happens to be optimized in the desired manner. This leaves the programmer dependent on the compiler to understand how to produce the desired idiom, and may require significant annotations to convince it that this is desirable. In some cases, this means that the desired output is simply impossible.

The ergonomic impact of paranoid practices is a tricky subject. For the most part, paranoid practices err on the side of producing false negatives – better to eliminate some good programs than allow bad programs to exist! This can lead to frustration if one wishes to produce a program they believe to be a false negative.

Omissions are generally much more well-accepted in this regard, particularly when an alternative is provided that handles common cases. The absence of manual free is quite popular because freeing memory is just annoying book-keeping. Freeing memory has no inherent semantic value, it's just necessary to avoid exhausting system resources. The fact that garbage collection just magically frees memory for you is pretty popular, as evidenced by its dominance of the programming language landscape (C and C++ being the most notable exceptions). However some omissions can be too large to bear; few systems are designed in a pure-functional manner.

Static analysis is much more maligned, as the developers of Coverity found [16]. It's easy to forgive an omission, particularly if one isn't even aware that an omission occurred (how many programmers are even aware of goto these days?). Static analysis, on the other hand, is active and in the programmer's face. It continuously nags the programmer to do a better job, or even prevents the program from running. Static typing, one of the less controversial strategies, is rejected by many popular languages (JavaScript, Python, Ruby). Meanwhile, dependent types struggle to find any mainstream usage at all. These practices can also harm the ability to continuously transform a program, forcing the programmer to prove that *everything* is correct in order to test *anything*.

Regardless, embracing paranoia *can* be liberating. Once a paranoid practice is in place, one never needs to worry about the problems it addresses again. Once you start collecting your own rainwater, why worry about what's being put in the water supply?

3.3 suspicion

Finally we have the position of compromise: the suspicious practices. A suspicious design often looks and acts a lot like a trusting design, but with one caveat: it actually checks at runtime. The suspicious stance is that some baseline level of safety is important, but categorically eliminating an error with a paranoid practice is impractical. The suspicious might give everyone guns, but make them explode whenever a HumanPointerException is detected.

Suspicious practices can skew to be implicit or explicit, in an attempt to fine-tune the tradeoffs.

In languages where pointers are nullable by default, it's common for all dereferences to silently and implicitly check for null. This may cause the program to completely crash or trigger unwinding (throw an exception). This allows the programmer to write the program in a trusting way without worrying about the associated danger.

Other interfaces may require the author to explicitly acknowledge invalid cases. For instance, an API may only provide access to its data through callbacks which it invokes when the data is specifically correct or incorrect. This approach is often more general, as it can be wrapped to provide the implicit form. Compared to implicit checks, explicit checks give more control to the user and can improve the clarity of code by making failure conditions more clear. But this comes at a significant ergonomic cost; imagine if every pointer dereference in Java required an error callback!

Regardless of the approach, suspicious practices are a decent compromise. For many problems the suspicious solution is much easier to implement, work with, and understand than the paranoid solution. Array indexing is perhaps the best example of this. It's incredibly simple to have array indexing unconditionally check the input against the length of the array. Meanwhile, the obvious paranoid solution to this problem requires an integer theorem solver and may require significant programmer annotations for complex access patterns. That said, paranoid solutions can also be simpler. Why check every dereference for null, when one can simply not let pointers be null in the first place?

One significant drawback of suspicious practices, particularly of the implicit variety, is that they do little for program *correctness*. They just make sure that the program doesn't do arbitrary unpredictable things with bad data, usually by reliably crashing the program. This can be great for detecting, reproducing, and fixing bugs, but a crash is still a crash to the end user. This can be especially problematic for critical systems, where a crash is essentially a vulnerability.

Suspicious solutions also often have the worst-in-class performance properties of the three solutions. Trusting solutions let the programmer do whatever they please while still aggressively optimizing. Paranoid solutions give the compiler and programmer rich information to optimize the program with. But suspicious solutions get to assume little, and must bloat up the program with runtime checks which further inhibit other optimizations. That said, a good optimizing compiler can completely eliminate this gap in some cases. Bounds checks in particular are easily eliminated in common cases [19].

4 RUST

Rust tries to be a practical language. It understands that each perspective has advantages and drawbacks, and tries to pick the best tool for each job. From each perspective, Rust takes one key insight. From the naive perspective, we embrace that control is incredibly important. Rust wants to be used in all the places where C and C++ are used today, and being able to manually manage memory is a critical aspect of that. From the paranoid perspective, we embrace that completely eliminating errors is great. Humans are fallible and get tired, but a good compiler never rests and always has our back. From the suspicious perspective, we embrace that compromises must be made for ergonomics.

Part of Rust's solution is then simply taking our favourite solutions for each specific problem: static types, runtime bounds checks, no nulls, wrapping arithmetic, and so on. However Rust has one very large holistic solution to many problems, and it's what separates it from most other languages: ownership. Rust's ownership model has two major aspects: controlling where and when data lives; and controlling where and when mutation can occur. These aspects are governed by three major features: affine types, regions, and privacy.

4.1 AFFINE TYPES

At a base level, Rust manages data ownership with an *affine type system*. The literature often describes affine types as being usable *at most once*, but from the perspective of ownership, affine typing means values are *uniquely owned*. To C++ developers, affine typing can be understood as a stricter version of *move semantics*.

If a variable stores a collection, passing it to a function by-value or assigning it to another variable transfers ownership of the value to the new location. The new location gains access to the value, and the old location loses access. Whoever owns the value knows that it's the only location in existence that can possibly talk about the contents of the collection. This allows the owner of the collection to soundly *trust* the collection. Any properties it observes and wishes to rely on will not be changed by some other piece of code without its permission. Perhaps more importantly, the owner of the collection knows that it can do whatever it pleases with the collection without interfering with anyone else.

A simple example:

```
fn main() {
    // A growable array
    let data = Vec::new();

    // transfer ownership of ‘data‘ to ‘data2‘
    let data2 = data;
    // ‘data‘ is now statically inaccessible,
    // and logically uninitialized

    // transfer ownership of ‘data2‘ to ‘consume‘
    consume(data2)
    // ‘data2‘ is now statically inaccessible,
    // and logically uninitialized
}

fn consume(mut data3: Vec<u32>) {
    // Mutating the collection is known to be safe, because
    // ‘data3‘ knows it’s the only one who can access it.
    data3.push(1);
}
```

The greatest of these rights is destruction: when a variable goes out of scope, it destroys its value forever. This can mean simply forgetting the value, or it can mean executing the type’s destructor. In the case of a collection, this would presumably recursively destroy all contained values, and free all of its allocations.

Affine types are primarily useful for eliminating the *use-after* family of bugs. If only one location ever has access to a value, and a value is only invalidated when that one location disappears, then it’s trivially true that one cannot use an invalidated value. For this reason, the most obvious applications of affine typing are with various forms of transient resources: threads, connections, files, allocations, and so on.

However it turns out that a surprising number of problems can be reduced to a use-after problem. For instance, many APIs require some sequence of steps to be executed in a certain order. This can be encoded quite easily using affine types. Functions can produce a “proof of work” by returning a type that only they have permission to produce. Similarly, functions can *require* a proof of work by consuming such a type:

```
fn first() -> First;
fn second(First) -> Second;
fn third(Second) -> Third;
fn alternative(First) -> Alternative;
```

We can therefore use affine types to model valid control flow and statically ensure correct usage. *Session types* are the logical extreme of this technique, where programs effectively "write themselves" due to their type constraints. Munksgaard and Jespersen [21] have an excellent analysis of session typing in Rust, so we won't dwell on this topic.

It should be noted that affine typing isn't mandatory in Rust. Unique ownership doesn't make sense or simply isn't important for many types like booleans and integers. Such types can opt into *copy semantics*. Copy types behave like any other value with one simple caveat: when they're moved, the old copy of the value is still valid.

Copy semantics can have surprising consequences though. For instance, it may be reasonable for a random number generator to be copyable, as its internal state is generally just some integers. It then becomes possible to *accidentally* copy the generator, causing the same number to be yielded repeatedly. For this reason, some types which *could* be copied safely don't opt into copy semantics. In this case, affine typing is used as a lint against what is likely, but not necessarily, a mistake.

4.2 BORROWS AND REGIONS

Affine types are all fine and good for some problems, but if that's all Rust had, it would be a huge pain in the neck. In particular, it's very common to want to *borrow* a value. In the case of a unique borrow, affine types can encode this fine: you simply pass the borrowed value in, and then return it back. This is *borrow threading*.

Threading is, at its best, just annoying to do. In particular, in must be written out in the types, and performed explicitly in the code. With only affine types, any process that borrows some data and has an actual return value requires all of the data to be mixed in with the return value. Say we'd like to write something like:

```
fn main() {
    let input = get_input();
    let pattern = get_pattern();
    if matches(&input, &pattern) {
        println!("input {} matches {}", input, pattern);
    }
}

fn matches(input: &Data, pattern: &Pattern) -> bool {
    // ...
    return found_match;
}
```

with only affine types we'd get something like:

```
fn main() {
    let input = get_input();
    let pattern = get_pattern();

    // Need to recapture all the data we loaned
    let (matches, input, pattern) = matches(input, pattern);

    if matches {
        println!("input {} matches {}", input, pattern);
    }
}

fn matches(input: Data, pattern: Pattern)
    -> (bool, Data, Pattern)
{
    // ...
    return (found_match, input, pattern);
}
```

Affine types *really* hit a wall when data wants to be *shared*. If several pieces of code wish to concurrently read some data, we have a serious issue. One solution is to simply *copy* the data to all the consumers. If each has their own unique copy to work with, everyone's happy.

However, even if we're ignoring the performance aspect of this strategy (which is non-trivial), it may simply not make sense. If the underlying resource to share is truly affine, then there may be *no* way to copy the data in a semantic-preserving way. For instance, one cannot just blindly copy a file handle, as each holder of the handle could then close it while the others are trying to use it.

At the end of the day, having only values everywhere is just dang *impractical*. Rust is a practical language, so it uses a tried and true solution: pointers! Unfortunately, pointers make everything more complicated and difficult. Affine types "solved" use-after errors for us, but pointers bring them back and make them *far* worse. The fact that data has been moved or destroyed says nothing of the state of pointers to it. As C has demonstrated since its inception, pointers are all too happy to let us view data that might be destroyed or otherwise invalid.

Garbage collection solves this problem for allocations, but does nothing to prevent trying to use an otherwise invalidated value, such as a closed file. Rust's solution to this problem is its most exotic tool: regions [23].

Like affine types, regions are something well-established in both theory and implementation, but with little mainstream usage. Although Rust primarily cribs them from Cyclone, they were first described by Tofte and Talpin [24] and used in MLKit. That said, Cyclone's version of regions is most immediately recognizable to a Rust programmer.

The idea of a region system is that pointers are associated with the region of the program that they're valid for, and the compiler ensures that pointers don't escape their region. This is done entirely at compile time, and has no runtime component.

For Rust, these regions correspond to lexical scopes, which are roughly pairs of matching braces. The restriction to lexical scopes is not fundamental, and was simply easier to implement for the 1.0 release. It is however sufficient for most purposes. Rust calls these regions *lifetimes*.

At a base level, all a region system does is statically track what pointers are outstanding during any given piece of code. By combining this information with other static analysis it's possible to completely eliminate several classes of error that are traditionally relegated to garbage collection. For ownership, region analysis allows us to statically identify when a value is moved or destroyed while being pointed to, and produce an error to that effect:

```
fn main() {
    // Gets a dangling pointer
    let data = compute_it(&0);
    println!("{}", data);
}

fn compute_it(input: &u32) -> &u32 {
    let data = input + 1;
    // Returning a pointer to a local variable
    return &data;
}
```

```
<anon>:11:13: 11:17 error: ‘data‘ does not live long enough
<anon>:11     return &data;
                      ^~~~
<anon>:8:36: 12:2 note: reference must be valid for the
    anonymous lifetime #1 defined on the block at 8:35...
<anon>: 8 fn compute_it(input: &u32) -> &u32 {
<anon>: 9     let data = input + 1;
<anon>:10     // Returning a pointer to a local variable
<anon>:11     return &data;
<anon>:12 }
<anon>:9:26: 12:2 note: ...but borrowed value is only valid
    for the block suffix following statement 0 at 9:25
<anon>: 9     let data = input + 1;
<anon>:10     // Returning a pointer to a local variable
<anon>:11     return &data;
<anon>:12 }
```

On its own, this is pretty great: no dangling pointers without the need for garbage collection! But when combined with affine types, we get something even more powerful than garbage collection. For instance, if you close a file in a garbage collected language, there is nothing to prevent old users of the file from continuing to work with it. One must guard for this at runtime. In Rust, this is simply not a concern: it's statically impossible. Closing the file destroys it, and that means all pointers must be gone. The problem of use-afters is once again solved.

Unfortunately, this doesn't solve problems like iterator invalidation. When an iterator is invalidated, the collection it was pointing to wasn't *destroyed*, it was just changed. In order to handle iterator invalidation, we require something more than checking for moves.

The most extreme solution is to simply forbid internal pointers. Only allow pointers to borrow variables on the stack, and everything else has to be copied out. Then we never have to worry about pointers being invalidated. Unfortunately, this would be a very limiting system. It would make composition of affine types useless, because you could never access the components without destroying the aggregate. It also doesn't really solve the problem the way we wanted. Most iterators we'd be interested in providing would become inexpressible under this model. For instance, one couldn't yield interior pointers from an iterator. Depending on the details, any kind of tree iterator may be completely impractical.

Another extreme solution would be to forbid mutation of data. Mutations can be emulated by creating a new object with the necessary changes, so this is in principle possible. However this suffers from similar issues as borrow threading: It's really annoying, and would also be make it difficult to obtain the same control as C(++).

Yet another way is to treat all pointers into a collection as pointers *to* the collection, and forbid mutation through pointers. All mutating operations could require by-value (and therefore unique) access, which could be done with borrow-threading. This is unfortunate because we were trying to avoid borrow-threading by introducing pointers in the first place, but at least we could share data immutably, which is a definite win.

Rust basically takes this last approach, but in order to avoid the annoying pain of threading borrows, it includes two different *kinds* of pointer: *mutable references* and *shared references* denoted `&mut` and `&` respectively. Shared references are exactly as we described: they can be freely aliased, but only allow you to read the data they point to. On the other hand, mutable must be unique, but enable mutation of the data they point to. This means that taking a mutable reference to some data is like moving it, but then having the compiler automatically insert all the boiler-plate to move it back into place when the mutable reference is gone (the compiler does not actually move the data around when you take a mutable reference).

Let's look at some simple examples:

```
let mut data = 0;

{
    // Allowed to take multiple shared references
    let data_ref1 = &data;
    let data_ref2 = &data;

    // Allowed to read through them,
    // and still read the value directly
    println!("{} {} {}", data_ref1, data_ref2, data);

    // Not allowed to mutate through them (compiler error)
    // *data_ref1 += 1;
}

{
    // Allowed to take one mutable reference
    let data_mut = &mut data;

    // Allowed to read or write through it
    println!("{}", data_mut);
    *data_mut += 1;

    // Allowed to move the mutable reference to someone else
    data_the_second = data_mut;

    // Not allowed to get an aliasing shared reference
    // (compiler error)
    // let data_ref = &data;

    // Not allowed to get an aliasing mutable reference
    // (compiler error)
    // let data_mut2 = &mut data;

    // Not allowed to directly access data anymore
    // (compiler error)
    // println!("{}", data);

    // But can get use the new location fine
    println!("{}", data_the_second);
}

// All borrows out of scope, allowed to access data again
data += 1;
println!("{}", data);
```

4.3 MUTABLE XOR SHARED

This is Rust's most critical perspective on ownership: mutation is mutually exclusive with sharing. In order to get the most out of this perspective, Rust doesn't allow mutability to be declared at the type level. That is, a struct's field cannot be declared to be constant. Instead, the mutability of a value is *inherited* from how it's accessed: as long as you have something by-value or by-mutable-reference, you can mutate it.

This stands in contrast to the perspective that mutation is something to be avoided completely. As we've seen, mutation can cause serious problems. This has lead some to conclude that mutation should be avoided as much as possible. Never mutating anything does indeed satisfy Rust's requirement that sharing and mutating be exclusive, but in a vacuous way (mutating never occurs). Rust takes a more permissive stance: mutate all you want as long as you're not sharing. The Rust developers believe that this eliminates most of the problems that mutation causes in practice.

In particular, this statically eliminates iterator invalidation. For instance, consider the following program:

```
fn main() {
    let mut data = vec![1, 2, 3, 4, 5, 6];
    for x in &data {
        data.push(2 * x);
    }
}
```

What exactly the programmer intended here was unclear, and what exactly will happen if this were allowed to compile is even more unclear. Thankfully, in Rust we don't *need* to wonder what the programmer meant or what will happen when this is run, because it doesn't compile:

```
<anon>:4:9: 4:13 error: cannot borrow ‘data‘ as mutable
    because it is also borrowed as immutable
<anon>:4         data.push(2 * x);
                 ^~~~
<anon>:3:15: 3:19 note: previous borrow of ‘data‘ occurs
    here; the immutable borrow prevents subsequent moves
    or mutable borrows of ‘data‘ until the borrow ends
<anon>:3     for x in &data {
                       ^~~~
<anon>:5:6: 5:6 note: previous borrow ends here
<anon>:3     for x in &data {
<anon>:4         data.push(2 * x);
<anon>:5     }
             ^
```

This strategy also nicely generalizes to a concurrent context. Recall that a data race is defined to occur when two threads access a piece of data in an unsynchronized way, and one is writing. This is exactly aliasing and mutation, which is forbidden by Rust's scheme. As such, everything in Rust is thread-safe by default.

Of course, perfectly good concurrent algorithms and data structures are rife with aliasing and mutability. Mutexes exist *precisely* to enable aliasing and mutation in a controlled manner. As a result, although inherited mutability is the default way to do things in Rust, it is not the only way. A few key types provide *interior mutability*, which enables their data to be mutated through shared references as long as some runtime mechanism ensures that access is properly restricted. The most obvious example of this is exactly the standard library's Mutex type, which allows an `&Mutex<T>` to become an `&mut T` by acquiring its lock:

```rust
use std::sync::Mutex;

fn main() {
    // A Mutex owns the data it guards. In this case,
    // an integer. Note that ‘data‘ is not declared
    // to be mutable, which normally would make it
    // impossible to update the value of the integer.
    let data = Mutex::new(0);

    {
        // Acquire the lock
        let mut handle = data.lock().unwrap();
        // A handle behaves like an ‘&mut‘ to the data
        *handle += 1;
        // But when it goes out of scope here,
        // it releases the lock.
    }
}
```

Why is this interface sound? First and foremost, any attempt to acquire the lock will block if it's already acquired. This ensures that only one handle exists at any given time. However, we must also guarantee that the handle doesn't outlive the mutex, and the pointer we get out of the handle doesn't outlive the handle. These problems are handled by ownership. Affinity and region analysis ensures that the pointers and handles aren't duplicated or allowed to outlive the type they refer to.

However the entire reason we care about Mutexes is for sharing across threads. This means there is one additional problem we must worry about: the shared data not being thread-safe. As we have noted, almost everything in Rust is actually thread-safe by default precisely because of ownership, but two things can potentially break this: borrows, and interior mutability. Borrows aren't trivially safe to share between two threads because they're based around sequential scopes. They don't really make sense with concurrent executions. Interior mutability isn't thread-safe because it's precisely sharing and mutation.

A Mutex provides interior mutability in an inherently thread-safe way, but not all types do. In particular, the Cell and RefCell types *aren't* thread-safe. So how do we ensure that these problematic types aren't shared across threads? For borrows, there's actually a way to declare that a type is expected to not contain borrows, so anything that can pass data to another thread requires that. However the interior mutability problem requires a completely different solution: Traits.

Traits are Rust's version of an interface. Rust actually captures thread-safety as traits that types can implement, called Send and Sync. If a type can be moved to another thread safely, then it is Send. If a type can be shared between two threads safely, then it is Sync. These traits are automatically derived compositionally; if you consist entirely of Send types, then you are Send. This works because of affinity and ownership. We know that if we own something that is thread-safe, then we are the only ones who can access it. So if we're accessed in a thread-safe way, then it's accessed in a thread-safe way.

Very few types are thread-unsafe, so almost everything is Send and Sync. However some types are specifically thread-safe even though they're based on parts that aren't. For instance, Mutex itself is based on parts that aren't thread-safe, but it is of course thread-safe as long as it contains thread-safe data. As such, types can manually claim to be Send or Sync. Of course, it's possible to make this claim incorrectly, so how is this safe to expose?

It's not safe. In fact, it's explicitly unsafe to implement these interfaces.

4.4 UNSAFE RUST

Most languages are considered memory-safe. However with few exceptions, this isn't actually true. In fact, basically *every* language has unsafe bits. The most fundamental of these is quite simple: talking to C. C is the lingua-franca of the programming world. All major operating systems and many major libraries primarily expose a C interface. Any language that wants to integrate with these systems must therefore learn how to talk to C. Because C is *definitely* unsafe and can do just about anything to your program, these languages then become transitively unsafe.

Rust is no different, but it embraces this reality a little more than most other languages. Rust is actually *two* languages: Safe Rust, and Unsafe Rust. Safe Rust is the Rust we have been focusing on for the most part. It is intended to be completely safe with one exception: it can talk to Unsafe Rust. Unsafe Rust, on the other hand, is definitely not a safe language. In addition to being able to talk to C (like any safe language), it enables the programmer to work with several constructs that would be easily unsound in Safe Rust. Most notably, for us, it allows Send and Sync to be implemented. However Unsafe Rust is most commonly used because it includes raw C-like pointers which are nullable and untracked.

At first glance, Unsafe Rust appears to completely undermine Rust's claims about safety, but we argue that it in fact *improves* its safety story. In most safe languages, if one needs to do something very low level (for performance, correctness, or any other reason) the general solution to this is "use C". This has several downsides.

First, there's a cognitive overhead. Such an application now has its logic spread across two completely different languages with different semantics, runtimes, and behaviors. If the safe language is what a development team primarily works in, it's unlikely that a significant percentage of the team is qualified to actively maintain the C components. Second, it incurs non-trivial runtime overhead. Data must often be reformatted at the language boundary, and this boundary is usually an opaque box for either language's optimizer. Finally, falling back to C is simply a *huge* jump in unsafety, from "totally safe" to "pervasively unsafe".

Unsafe Rust largely avoids these issues with one simple fact: it's just a superset of Safe Rust. Lifetimes, Affine Types, and everything else that helps you write good Rust programs are still working exactly as before. You're just allowed to do a few extra things that are unsafe. As a result, there's no unnecessary runtime or semantic overhead for using Unsafe Rust.

Of course one *does* need to understand how to manually uphold Safe Rust's various guarantees when using Unsafe Rust's extra parts, and this isn't trivial. However this is still a better situation than using C, because the unsafety is generally much more modular. For instance, if you use Unsafe Rust to index into an array in an unchecked manner, you don't suddenly need to worry about the array being null, dangling, or containing uninitialized memory. All you need to worry about is if the index is actually in bounds. You know everything else is still normal.

In addition, Unsafe Rust doesn't require any kind of complicated foreign function interface. It can be written inline with Safe Rust on demand. Rust's only requirement is that you write the word "unsafe" *somewhere* to indicate that you understand that what you're doing is unsafe. Since unsafety is explicitly denoted in this manner, it also enables it to be detected and linted against if desired.

Rust's standard library (which is written entirely in Rust) makes copious use of Unsafe Rust internally. Most fundamentally, Unsafe Rust is necessary to provide various operating system APIs because those are written in C, and only Unsafe Rust can talk to C. However Unsafe Rust is also used in various places to implement core abstractions like mutexes and growable arrays.

It's important to note that the fact that these APIs use unsafe code is entirely an implementation detail to those using the standard library. All the unsafety is wrapped up in *safe abstractions*. These abstractions serve two masters: the consumer of the API, and the producer of the API. The benefit to consumers of an API is fairly straight-forward: they can rest easy knowing that if something terrible happens, it wasn't their fault. For producers of the API, these safe abstractions mark a clear boundary for the unsafety they need to worry about.

Unsafe code can be quite difficult because it often relies on stateful invariants. For instance, the capacity of a growable array is a piece of state that unsafe code must trust. In order to be sound, these safe abstractions need to rely on the final element of ownership: *privacy*.

Privacy in Rust is much the same as in most other languages. Fields and functions may be marked as public or private, and only code that is within some boundary may access anything that is marked private.

Returning to our example, the capacity of a growable array is marked as private. Since the abstraction boundary is often exactly the privacy boundary in Rust, end users of a growable array are therefore prevented from directly manipulating the capacity. Within the array's privacy boundary, this state can be arbitrarily manipulated, but this is a closed set of code to audit and verify. The code within the privacy boundary can therefore trust that the capacity field is only updated by a small set of trusted code.

This rest of this thesis focuses primarily on these safe abstractions. A good safe abstraction must have many properties:

1. Safety: Using the abstraction inappropriately cannot violate Rust's safety guarantees.

2. Efficiency: Ideally, an abstraction is *zero cost*, meaning it is as efficient at the task it is designed to solve as an unabstracted solution (with a decent optimizing compiler).

3. Usability: A good abstraction should be more convenient and easy to understand than the code it's wrapping.

It would be *excellent* if the implementation was also completely safe, but we do not consider this a critical requirement, as Rust's standard library demonstrates.

It should be noted that Rust's reliance on safe abstractions is, in some sense, unfortunate. For one, it makes reasoning about the performance characteristics of a program much more difficult, as it relies on a sufficiently smart compiler to tear away these abstractions. This in turn means Rust's unoptimized performance is in a rather atrocious state. It's not uncommon for a newcomer to the language to express shock that a Rust program is several times slower than an equivalent Python program, only to learn that enabling optimizations makes the Rust program several times *faster* than the Python program (and indeed, as fast as one would expect from the equivalent C++). However it is our opinion that this is simply fundamental to providing a programming environment that is safe, efficient, and usable.

5 DESIGNING APIS FOR TRUST

It has been our experience that almost everything you want to express at a high level can be safely, efficiently, and usably expressed in an ownership-oriented system. However this does *not* mean that you can express it however you please! In order to be safe, efficient, and usable the API itself must be designed in terms of data trust. If the API isn't designed with trust and ownership in mind, these goals will likely be compromised.

We have already seen how ownership eliminates use-afters, view invalidation, and data races. However we have not yet seen how indexing is addressed.

5.1 INDEXING

Arrays are overwhelmingly the most common and important data structure in all of programming. Basically every program will do some array processing – even *hello world* is just copying an array of bytes to stdout. Array processing is so pervasive that Rust provides no less than *four* different ways to index into an array, each corresponding to one of the trust strategies: naive, paranoid, implicitly suspicious, and explicitly suspicious.

First and foremost, Rust provides the two most popular interfaces: completely unchecked indexing (via Unsafe Rust), and implicitly checked indexing (which unwinds the program if it fails). Nothing particularly surprising or novel there.

Slightly more interesting is how Rust provides an explicitly checked option. Explicit checking is most commonly done through the Option and Result types in Rust. If an operation can fail, it will return one of these types. These types are tagged unions, which means they can contain different types at runtime. 'Option<T>' can be 'Some(T)' or 'None', while 'Result<T, E>' can be 'Ok(T)' or 'Err(E)'. By owning the data they wrap, these types can control how it's accessed. Both of them provide a multitude of ways to access their contents in an explicitly checked way.

Explicitly checked indexing returns an Option, so if we want to explicitly handle the failure condition, we can do any of the following:

```
// Exhaustively match on the possible choices
match array.get(index) {
    Some(elem) => println!("Elem: {}", elem),
    None       => println!("Nothing"),
}
```

```
// Only handle one pattern
if let Some(elem) = array.get(index) {
    println!("Elem: {}", elem);
}

// Execute a callback if the value is Some
array.get(index).map(|elem| {
    println!("Elem: {}", elem);
});

// ... and more
```

It's worth noting that checked indexing is surprisingly performant here. First off, these bounds checks are by definition trivially predictable in a correct program. So the overhead at the hardware level is quite small. That said, having to do the checks at all is the worst-case. A good optimizing compiler (like LLVM) can optimize away bounds checks in many "obviously correct" cases. For instance, the following code doesn't actually perform any bounds checks when optimized, because LLVM can see that the way the indices are generated trivially satisfies the bounds checking.

```
let mut x = 0;
for i in 0 .. arr.len() {
    // implicit checked indexing
    x += arr[i];
}
```

Indeed, if you can convince LLVM to not completely inline and constant-fold this code away, it will even successfully vectorize the loop!

However compiler optimizations are brittle things that can break under even seemingly trivial transformations. For instance, changing this code to simply iterate over the array *backwards* completely broke LLVM's analysis and produced naive code that adds the integers one at a time with bounds checking. This is perhaps the most serious cost of bounds checks: inhibiting other optimizations.

If we really care about avoiding this cost, we can't just rely on the optimizer to magically figure out what we're doing, we need to actually not do bounds checking. We can use the unsafe unchecked indexing, but we'd rather not resort to unsafe code unless totally necessary. What we really want here is Rust's final solution to indexing: the paranoid one.

This code is hard to optimize safely because we've pushed too much of the problem at hand to the user of the array. They need to figure out how to generate the access pattern, and we in turn can't trust it. If the *array* handles generating the access pattern *and* acquiring the elements, then all the bounds checks can be eliminated at the source level in a way that's safe to the end-user. This is handled by a tried and true approach: iterators.

```
let mut sum = 0;
for x in arr.iter() {
    sum += *x;
}
```

This produces the same optimized code as the original indexing-based solution, but more importantly, it's more robust to transformations. Iterating backwards now also produces vectorized unchecked code, because the optimizer has less to prove about the program's behaviour. As an added bonus, client code ends up simplified as well, as all the error-prone iteration boilerplate has been eliminated.

Of course, this doesn't come for free. Iterators are effectively special-casing certain access patterns at the burden of the interface's implementor. In the case of iterators, this burden is completely justified. Linear iteration is incredibly common, and therefore well worth the specialization. But if the user wants to binary search an array without bounds checks, iterators do them no good.

This is why Rust also provides the raw unchecked indexing API. If users have a special case and really need to optimize it to death, they can drop down to the raw API and regain full control. In the case of arrays, the raw API is trivial to provide and obvious, so there's little worry of the maintenance burden or unnecessarily exposing implementation details. Unfortunately, this isn't true for all interfaces (what's the raw API for searching an ordered map?).

5.2 EXTERNAL AND INTERNAL INTERFACES

It is relatively common to use maps as *accumulators*. The most trivial example of this is using a map to count the number of occurrences of each key. Accumulators are interesting because special logic must usually be performed when a key is seen for the first time. In the case of counting keys, the first time we see a key we want to insert the value 1, but each subsequent time we see that key we want to instead increment the count.

Naively, one may write this as follows:

```
if map.contains(&key) {
    map[key] += 1;
} else {
    map.insert(key, 1);
}
```

Those concerned with performance may see an obvious problem with this implementation: we're unnecessarily looking up each key twice. Instead we would like to search in the map only once, and execute different logic depending on if the key was found or not *without* performing the search again.

Before Rust 1.0 was released, there existed a family of functions for doing exactly that. For the simple case of a counter, we only need to provide a default value, which works well enough:

```
*map.find_or_insert(key, 0) += 1;
```

However the default value might be expensive to create for some kinds of accumulators. As such, we'd like to avoid constructing it unless we know it's required. And so, `find_or_insert_with` was provided, which took a *function* that computed the default value:

```
*map.find_or_insert_with(key, expensive_default_func) += 1;
```

However this interface had the problem that it could be difficult to tell which case was found. This is where things started to fall apart. In order to support this, a function that took *two* functions was created; one for each case. However this design was problematic because each function may want to capture the same affine data by-value. We know this is sound because only one of the two functions will be called, but the compiler doesn't understand that. So an extra argument was added which would be passed to the function that *was* called.

```
map.update_with_or_insert_with(key, capture,
                               compute_default_func,
                               update_existing_func);
```

While the first APIs seemed quite reasonable, this final API was nightmarish. Worse, it didn't even accommodate all the use cases people came up with. Some wanted to *remove* the key from the map under some conditions, which would necessitate a whole new family of update functions. The problem is that this design is what the Rust community calls an *internal* interface. Internal interfaces require the client to execute *inside* the interface's own code by passing functions that should be called at the appropriate time. In some cases, internal interfaces can be convenient or even more efficient, but they generally suffer from how little control they give the client.

We solved this inflexibility by instead providing an *external* interface. Internal interfaces execute the entire algorithm at once, invoking the client to handle important cases. An external interface instead requires the client to drive the execution of the algorithm manually. At each step, the algorithm returns some value that summarizes the current state, and exposes relevant operations.

For the accumulator problem, we created the *entry* API. The basic idea of the entry API is simple: execute the insertion algorithm up until we determine whether the key already existed. Once this is known, take a *snapshot* of the algorithm state, and store it in a tagged union with two states: Vacant or Occupied. The consumer of the interface must then match on the union to determine which state the algorithm is in. The Vacant state exposes only one operation, insert, as this is the only valid operation to perform on an empty entry. The Occupied state, on the other hand, exposes the ability to overwrite, remove, or take a pointer to the old value.

In its most general form, usage looks as follows:

```
// Search for this key, and capture whether it's in
// the map or not
match map.entry(key) {
    Vacant(e) => {
        // The key is not in the map, compute the new value
        let value = expensive_default(capture);
        e.insert(value);
    }
    Occupied(e) => {
        // The key is in the map, update the value
        expensive_update(e.get_mut(), capture);

        // Conditionally remove the key from the map
        if *e.get() == 0 {
            e.remove();
        }
    }
}
```

Control flow is now driven by the client of the API, and not the API itself. No additional interfaces need to be added to accommodate all the different actions that are desired, and no additional lookups are performed.

Of course, this is a significant ergonomic regression for simple cases like counting, for which convenience methods were added to the Entry result:

```
*map.entry(key).or_insert(0) += 1;
```

One may question if we have then gained much if we're still adding some of the old interfaces this design was intended to replace, but there is an important difference. Before, we were required to add new interfaces to accommodate increasingly complex use cases. Now, we are adding new interfaces to accommodate increasingly *common* use cases. We believe this to be the more correct way for an interface to grow; adding conveniences for idioms, rather than adding nightmares for special cases.

One important question about this interface is whether it's *sound*. After all, we're taking a snapshot of the internal state of a collection, and then yielding control to the client explicitly to mutate the collection. Indeed, in many languages this interface would be dangerous without runtime checks for exactly the same reason that iterators are dangerous. And for exactly the same reason that iterators can safely be used without any runtime checks, so can entries: ownership!

An Entry mutably borrows the map it points into, which means that while it exists, we know that it has exclusive access to the map. It can trust that the algorithmic state it recorded will never become inconsistent. Further, any operation that the entry exposes that would invalidate itself (such as inserting into a vacant entry or removing from an occupied entry) consumes it by-value, preventing further use.

The comparison to iterators is particularly apt here because iterators are yet another external interface. They require the consumer to repeatedly request the next element, returning an Option that is None if the iteration is complete. Like entries, iterators in Rust were once provided as an internal interface. Iteration required a function to be passed to the iterator, which it would then execute on every element.

The most fundamental weakness of this design is that it was impossible to concurrently iterate over two sources at once. Each iterator could only be executed to completion at once. With external iterators, concurrent iteration is simply alternating which iterator to ask for the next element.

5.3 HACKING GENERATIVITY ONTO RUST

Given the array iteration example, one might wonder if it's sufficient for the array to simply provide the indices, and not immediately convert them into elements. This would be a more composable API with a reduced implementation burden.

Perhaps this API could be used something like this:

```rust
let mut sum = 0;
for i in arr.indices() {
    sum += arr[i];
}
```

Unfortunately, this doesn't immediately get us anywhere. This is no different than the original example which produced its own iteration sequence. As soon as the array loses control of the yielded indices, they are *tainted* and all trust is lost. After all, they're just integers, and integers can come from anywhere. One may consider wrapping the integers in a new type that doesn't expose the values to anyone but the array, preventing them from being tampered with, but this is still insufficient unless the *origin* of these values can be verified. Given two arrays, it mustn't be possible to index one using the indices of the other:

```
let mut sum = 0;
for i in arr1.indices() {
    // index arr2 using arr1's trusted indices
    sum += arr2[i];
}
```

This is a problem many static type systems generally struggle to model, because they don't provide tools to talk about the origin or destination of a particular instance of a type. In addition, even if we could prevent this, we would have to deal with the problem of temporal validity. The indices into an array are only valid as long as the array's length doesn't change:

```
// get a trusted index into the array
let i = arr.indices().next().unwrap();
// shrink the array
arr.pop();
// index the array with an outdated index
let x = arr[i];
```

By pure accident, Rust provides enough tools to solve this problem. It turns out that lifetimes in conjunction with some other features are sufficient to introduce *generativity* into the type system. Generativity is a limited system that can solve some of the problems usually reserved for dependent typing. I won't lie: this is an ugly hack, and I don't expect it to see much use, although early drafts of this work have inspired the creation of at least one library [1]. Regardless, it demonstrates the power of the ownership system.

In order to encode sound unchecked indexing, we need a way for types to talk about particular instances of other types. In this case, we specifically need a way to talk about the relationship between a particular array, and the indices it has produced. Rust's lifetime system, it turns out, gives us exactly that. Every instance of a value that contains a lifetime (e.g. a pointer) is referring to some particular region of code. Further, code can require that two lifetimes satisfy a particular relationship. Typically all that is required is that one outlives the other, but it is possible to require strict equality.

The basic idea is then as follows: associate an array and its indices with a particular region of code. Unchecked indexing then simply requires that the input array and index are associated with the same region. However care must be taken, as Rust was not designed to support this. In particular, the borrow checker is a constraint solver that will attempt do everything in its power to make code type-check. As such, if it sees a constraint that two lifetimes must be equal, it may expand their scopes in order to satisfy this constraint. Since we're trying to explicitly use equality constraints to prevent certain programs from compiling, this puts us at odds with the compiler.

In order to accomplish our goal, we need to construct a black box that the borrow checker can't penetrate. This involves two steps: disabling lifetime variance, and creating an inaccessible scope. Disabling variance is relatively straight forward. Several generic types disable variance in order to keep Rust sound. Getting into the details of this is fairly involved, so we will just say we can wrap a pointer in the standard library's Cell type as follows:

```
struct Index<'id> {
    data: usize,
    _id: Cell<&'id u8>,
}
```

The 'id syntax is new here. Although most Rust programs can avoid declaring lifetimes, more advanced usage necessitates declaring them as generic arguments. In this case, we're externally declaring that the Index type contains something with a lifetime called id. Internally we're declaring this to be the lifetime of some pointer to a byte.

Of course we don't *actually* want to store a pointer at runtime, because we're only interested in creating a lifetime for the compiler to work with. Needing to signal that we contain a lifetime or type that we don't directly store is a sufficiently common requirement in Unsafe Rust that the language provides a primitive for exactly this: PhantomData. PhantomData tells the type system "pretend I contain this", while not actually taking up any space at runtime.

```
// Synonym to avoid writing this out a lot
type Id<'id> = PhantomData<Cell<&'id u8>>;

struct Index<'id> {
    data: usize,
    _id: Id<'id>,
}
```

Now Rust believes it's unsound to freely resize the id lifetime. However, as written, there's nothing that specifies where this id should come from. If we're not careful, Rust could *still* unify lifetimes incorrectly if it notices there's no actual constraints on them. Consider the following kind of program we're trying to prevent:

```
let arr1: Array<'a> = ...;
let arr2: Array<'b> = ...;
let index: Index<'a> = arr1.get_index();
```

```
// This should fail to compile;
// trying to index Array<'b> with Index<'a>
let x = arr2[index];
```

If we don't constrain the lifetimes a and b, then the constraint solver will see only the following system:

- a and b are invariant lifetimes
- indexing requires $a = b$

Which has the obvious solution of $a = b = anything$. We need to apply some constraint to a and b to prevent Rust from unifying them. Within a single function the compiler has perfect information and can't be tricked. However Rust explicitly does not perform inter-procedural analysis, so we can apply constraints with functions. In particular, Rust has to assume that the *input* to a function is a fresh lifetime, and can only unify them if that function provides constraints:

```
fn foo<'a, 'b>(x: &'a u8, y: &'b u8) {
    // cannot assume x has any relationship to y, since they
    // have their own lifetimes
}

fn bar<'a>(x: &'a u8, y: &'a u8) {
    // x has the same lifetime as y, since they share 'a
}
```

Therefore, for every fresh lifetime we wish to construct, we require a new function call. We can do this as ergonomically as possible (considering this is a hack) by using closures:

```
fn main() {
    let arr1 = &[1, 2, 3, 4, 5];
    let arr2 = &[10, 20, 30];

    // Yuck! So much nesting!
    make_id(arr1, move |arr1| {
    make_id(arr2, move |arr2| {
        // Within this closure, Rust is forced to assume
        // that the lifetimes associated with arr1 and
        // arr2 originate in their respective make_id
        // calls. As such, it is unable to unify them.

        // Iterate over arr1
        for i in arr1.indices() {
            // Will compile, no bounds checks
```

```
                println!("{}", arr1.get(i));

                // Won't compile
                println!("{}", arr2.get(i));
            }
        });
    });
}

// An Invariant Lifetime
type Id<'id> = PhantomData<Cell<&'id u8>>;

// A wrapper around an array that has a unique lifetime
struct Array<'arr, 'id> {
    array: &'arr [i32],
    _id: Id<'id>,
}

// A trusted in-bounds index to an Array
// with the same lifetime
struct Index<'id> {
    idx: usize,
    _id: Id<'id>,
}

// A trusted iterator of in-bounds indices into an Array
// with the same lifetime
struct Indices<'id> {
    min: usize,
    max: usize,
    _id: Id<'id>,
}

// Given a normal array, wrap it to have a unique lifetime
// and pass it to the given function
pub fn make_id<'arr, F>(array: &'arr [i32], func: F)
    where F: for<'id> FnOnce(Array<'arr, 'id>),
{
    let arr = Array { array: array, _id: PhantomData };
    func(arr);
}

impl<'arr, 'id> Array<'arr, 'id> {
    // Access the following index without bounds checking
    pub fn get(&self, idx: Index<'id>) -> &'arr i32 {
        unsafe { return self.array.get_unchecked(idx.idx); }
    }

    // Get an iterator over the indices of the array
    pub fn indices(&self) -> Indices<'id> {
        return Indices {
            min: 0,
            max: self.array.len(),
            _id: PhantomData
        };
    }
}

impl<'id> Iterator for Indices<'id> {
    type Item = Index<'id>;
    pub fn next(&mut self) -> Option<Self::Item> {
        if self.min == self.max {
            return None;
        } else {
            self.min += 1;
            return Some(Index {
                idx: self.min - 1,
                _id: PhantomData
            });
        }
    }
}
```

That's a *lot* of work to safely avoid bounds checks, and although there's only a single line marked as 'unsafe', its soundness relies on a pretty deep understanding of Rust. Any errors in the interface design could easily make the whole thing unsound. So we *really* don't recommend doing this. Also, whenever this interface catches an error, it produces nonsensical lifetime errors because Rust has no idea what we're doing. That said, it does demonstrate our ability to model particularly complex constraints.

Those familiar with generativity and type systems may see that we are ultimately just applying an age-old trick: *universal* types and functions can be combined to construct *existential* types. In this case `where F: for<'id> FnOnce(Array<'arr, 'id>)` is declaring that the function F is universal over all lifetimes that can be chosen for `id`. The body of any function that satisfies this signature must work with any `id` it receives opaquely. In effect, it knows that there *exists* a lifetime, and nothing else.

6 LIMITATIONS OF OWNERSHIP

We've seen that ownership allows us to cleanly model all of the major classes of error that we were interested in solving. This is because ownership's primary role is to prevent data from being accessed at inappropriate times, and this is what all of those problems boiled down to. However ownership is unable to properly model some problems and can severely limit some designs.

6.1 CYCLIC REFERENCES

Ownership is severely biased towards unique ownership. This implies that data is laid out in a tree-like manner, without any cycles. If one's problem can be modeled without referential cycles, then ownership is pleasant and easy to use. Otherwise, ownership becomes quite burdensome.

Any naive attempt to create a graph, tree with parent pointers, doubly-linked list, or a value that contains pointers to itself will quickly lead to the compiler shutting everything down. This is necessitated by the fact that Rust allows memory to be manually managed, but demands that pointers can never dangle. Anyone who has tried to build a pointer-based data structure knows all too well how easy it is to forget to update some of the pointers after a node is removed.

Therefore Rust refusing to compile this sort of code is *correct*. Pointer-based data structures are wildly unsafe without garbage collection, and are error-prone even *with* garbage collection. Still, they're useful, and people want to write them. Questions about this problem are so frequent that we wrote an entire book on the topic [13].

The most straight-forward solution to this problem is to embrace garbage collection. Although Rust doesn't provide any serious automatic tracing collector, it does provide manual reference counted pointers (Rc). Rc is Rust's primary mechanism for *shared ownership*. No Rc is the true owner of the pointed-to data. Instead, all Rcs have equal ownership. Because the data pointed to by an Rc is always potentially shared, only immutable shared access is provided. On its own, this significantly limits the usefulness of Rc – it's not possible to construct a cycle with strict evaluation and immutability!

The solution to this problem is interior mutability. With interior mutability and reference counting, one can safely build and manage cyclic data structures at the cost of reference counting and runtime ownership checking. This also comes at a significant ergonomic cost.

Compare these equivalent Rust and Java programs, which just make a cycle of two nodes:

```rust
use std::rc::Rc;
use std::cell::RefCell;

// Magical annotation to get a Node::default constructor
#[derive(Default)]
struct Node {
    data: u32,
    next: Option<Rc<RefCell<Node>>>,
    prev: Option<Rc<RefCell<Node>>>,
}

fn main() {
    let a = Rc::new(RefCell::new(Node::default()));

    let b = Rc::new(RefCell::new(Node {
        data: 1,
        next: Some(a.clone()),
        prev: Some(a.clone()),
    }));

    a.borrow_mut().next = Some(b.clone());
    a.borrow_mut().prev = Some(b.clone());
}
```

```java
class Main {
    public static void main (String[] args) {
        class Node {
            int data;
            Node next;
            Node prev;
        }

        Node a = new Node();

        Node b = new Node();
        b.data = 1;
        b.next = a;
        b.prev = a;

        a.next = b;
        a.prev = b;
    }
}
```

The Rust program is completely bogged down in boiler-plate to properly create the pointers, copy them, and update the contents. This is to some extent desirable as Rust likes to make costs explicit, and node-based data structures are quite heavyweight for low level programming. Defaulting to pervasive garbage collection and nullable everything ends up being quite nice for this use case.

The primary workaround for this problem is to use arrays and indices instead of pointers. For many workloads, this is completely sufficient, and even more efficient than a naive graph. As a rough example:

```rust
#[derive(Default)]
struct Node {
    data: u32,
    next: Option<usize>,
    prev: Option<usize>,
}

fn main() {
    let mut graph = Vec::new();
    graph.push(Node::default());
    graph.push(Node {
        data: 1,
        next: Some(0),
        prev: Some(0),
    });

    graph[0].next = Some(1);
    graph[0].prev = Some(1);
}
```

This is, in essence, completely abandoning the borrow checker. It only reasons about pointers, and we're using indices into the array. This allows us to completely invalidate nodes by removing the things they "point" to, and even perform what is logically equivalent to a use-after free by removing a node and then reinserting a node. However we're unable to violate memory safety in this manner, because the indices will be bounds checked. We'll just get nonsensical results, which is of course quite easy to do with any node-based structure.

The petgraph library [6] provides a more complete interface for building and working with graphs in Rust. Internally, it just boils down to this basic design. Graphs are an array of nodes and an array of edges, and nodes and edges just store indices into those arrays. This structure is used by the Rust compiler itself (which is written entirely in Rust).

Of course, this is only necessary if one wants to implement cyclic structures safely. It's entirely possible to implement them exactly as they would be in C in all their unsafe glory. This is how Rust's LinkedList and BTreeMap collections are implemented. As usual, the unsafety is easily hidden behind a reasonable and efficient public interface.

6.2 LEAKS

Leaks aren't well-modeled by Rust's ownership system because it's based on affine types, which allow us to forget about values at any time. In order to encode that a value must be properly consumed, we need to be able to express that a value must be used *exactly* once. Types with this property are said to be *linear*.

A poor-man's linear-typing can be acquired with *destructors*. A value's destructor is some code to execute when it goes out of scope. This effectively mandates that values with destructors be used exactly once: the owner who decides to drop the value on the ground is forced to invoke the value's destructor.

The most obvious limitation to this approach is that since destructors are implicitly invoked, they cannot take any additional context. For instance, it may be desirable to have a type that is allocated using a custom allocator, but does not store a pointer to that allocator. The type therefore has insufficient information to clean itself up, requiring the allocator be passed to it. A more robust linearity system would allow us to encode this by requiring the destructor be explicitly invoked with the allocator as an argument. Destructors also can't return anything to indicate success or failure, which is problematic for the automatic closing of files.

That said, destructors do a great job for many types. Collections and connections have a natural final operation that requires no additional context. Collections free their allocations and connections close themselves. Destructors also have the distinct advantage over more general linearity in that they compose well with generic code. If some generic code wants to drop a value on the ground, it can always do this knowing any linear requirements will be satisfied by the destructor.

Unfortunately, ensuring that destructors execute is a nasty problem. At the limit, hardware can fail and programs can abort. We just need to live with that fact. For most resources this is actually fine; either the resources are rendered irrelevant by the program ending, or the operating system will automatically clean them all up itself.

Even accepting those circumstances, strict linearity can be burdensome. In particular, it is desirable to be able to create reference-counted cycles of types that have destructors. If all references to such a cycle are lost it will keep itself alive, leaking the destructors it owns. Further, it is sometimes desirable to manually prevent a destructor from running, particularly when decomposing a value into its constituent parts. For instance, one may wish to downgrade a pointer that would normally free its allocation to a raw pointer, perhaps to pass to a C API.

There are various solutions to this problem, but most languages that include destructors (C++ C#, Java, D, ...) generally accept that sometimes they won't run. They're convenient and generally reliable, but if one *really* needs something to happen, they can't always be relied on. Rust also takes this stance. In particular, one can't pass a value with a destructor to an arbitrary third-party and rely on the destructor to be called, even if the compiler believes any borrows they held have expired.

It should however be noted that this is *completely* a library decision, and not a language one. Rust's standard library decided that leaking destructors was a safe operation, and safe interfaces need to assume it can happen. However there exists an alternative standard library that adopted a different stance, making this an unsafe operation [7]. We won't dwell on the details, but the basic idea is to encode leak-safety as an unsafe trait like the standard library does for thread-safety.

As a concrete example of dealing with the ability to have a destructor leaked, we can consider Rust's *drain* interface. Drain is a utility for efficiently performing bulk removal from the middle of a growable array. Arrays require their elements to be stored contiguously, so removing from the middle of one generally requires the elements after it to be shifted back to fill the hole. Doing this repeatedly is incredibly expensive, so it's desirable to be able to defer the shift until we are done removing elements. Doing this means the array is in an unsound state while the removals are happening, and we don't want this to be observable by the client.

One solution to this problem is to require the caller to pass in a target buffer for all the elements that will be removed. Then 'drain' can just run to completion without ever yielding control to the client. Unfortunately, this is quite inflexible.

To get more control we once again created an external interface, implementing 'drain' as an iterator. Elements are removed one at a time, allowing the user to decide what is done with them without any need for allocating a buffer. At first blush, the safety of this interface seems trivially solved by ownership. Drain is mutating the array, so it contains a mutable reference to the array. That means the array is inaccessible through any interface but the iterator while it exists. The shifting can then be done in the iterator's destructor, which is exactly when the borrow it holds expires. Perfect!

```
// A growable array with 5 numbers
let mut arr = vec![0, 1, 2, 3, 4];

// Drain out elements 1 to 3 (exclusive).
// The result of 'drain' is an iterator
// that we pass to the for loop to drive.
for x in arr.drain(1..3) {
    // arr is statically inaccessible here
    println!("{}", x);
}
// backshifting is performed here
// arr is now accessible again

// 1 and 2 are now gone
assert_eq!(arr, vec![0, 3, 4]);
```

Unfortunately, the soundness of this design relies on the user of our interface allowing the destructor to run. No reasonable code would ever *not* run the destructor, but since one can violate memory safety by accessing these empty indices, we cannot tolerate the possibility.

Thankfully, all is not lost. The solution to our problem is in fact quite simple. Rather than relying on the destructor to put the array in a sound state, we will *start* the drain by putting the array into an incorrect but otherwise sound state, and rely on the destructor to put the array into the correct state. If the user of our interface prevents the destructor from running, they'll get the incorrect array which will probably cause their program to behave incorrectly, but be unable to violate memory safety. Since no reasonable program will ever trigger this, the possibility of incorrect state isn't particularly worrying.

In the case of 'drain', the incorrect state is setting the length of the array to be zero. The destructor in turn just sets the length to the correct value (which it would have done anyway). So the overhead for this scheme is zeroing out a single integer, which is completely negligible considering the operation we're performing. The consequence of this is that all the elements in the array will be lost if the destructor is leaked. There's a certain fairness to this: *you leak me, I leak you!* We call this *leak amplification*.

Not all interfaces have such a convenient state though. If this is the case, one may instead ensure some code executes by using an internal interface. Rather than returning an object that can be forgotten, we can require the user to pass a function that will be passed a *pointer* to the object, preventing them from gaining true ownership of it. We can then guarantee that the object's destructor always runs by dropping it in our own function. In fact, because we always control the value, we don't even need to use a destructor to ensure the cleanup code runs. We can just have that code at the end of the function.

6.3 SCOPED THREADS

With that said, we finally have all the tools to understand the interface we demonstrated at the start of this thesis. Scoped threads allow data to be borrowed by another thread, and even mutated. As we discussed previously, this is quite dangerous, because borrows don't understand concurrency. So how does this work? The initial scoped thread interface was designed to work as follows:

```
use std::thread;

fn main() {
    // An array of integers, stored on the stack
    let mut data = [1, 2, 3];

    {
        // An array of thread join guards. We will destroy this
        // when we want to block on the threads completing.
        let mut guards = Vec::new();

        // Get pointers to each element in the array
        for x in &mut array {
            // Spawn a thread to increment this element
            let guard = thread::scoped(move || {
                *x += 1;
            });
            guards.push(guard);
        }

        // guards go out of scope here, so we block
        // on all the child threads joining.
    }

    // Array is now done being modified, and safe to read
    println!("{:?}", array);
}
```

The interesting piece of this API was the guards that were returned by the scoped function. Each guard claimed to store the closure the scoped function was passed. This in effect made them borrow all of the data that the closure could access. So as long as the guard existed, nothing could interfere with data that the spawned threads were accessing. No magic here, just the borrow checker doing its job.

However the borrows expire when the guard goes away, so the guard needs to make sure that the spawned thread is no longer running. The only way to do that is to wait for the thread to finish, and that's exactly what a guard's destructor did. But we have a problem: leaking the destructor. If the destructor is prevented from running, the borrow checker will believe all borrows have expired, and allow the array to be accessed while it's still potentially being mutated on another thread.

Unfortunately for the scoped interface, there's no valid leak amplification strategy. We would have to somehow invalidate all the data that was borrowed, but that's not possible. So we must use an internal interface, resulting in the example we saw in the introduction:

```
[dependencies]
crossbeam = 0.1.6

-----

extern crate crossbeam;

fn main() {
    let mut array = [1, 2, 3];

    // scope is essentially an ‘&mut Vec<Guard>‘
    crossbeam::scope(|scope| {
        for x in &mut array {
            // scope.spawn is essentially guards.push(guard)
            scope.spawn(move || {
                *x += 1;
            });
        }
    });

    println!("{:?}", array);
}
```

This design is a bit less flexible. In particular, one loses control of how the guards are stored and when they are destroyed. A more complicated design could in principle expose that functionality, but we haven't yet found the need to do so.

Regardless, the end result is impressive: in spite of the limitations of ownership being a fairly simple and limited system, we can use it to safely share data stored on the stack of one thread with several other threads without unnecessary synchronization!

7 RELATED WORK

Since Rust's model of ownership is built up of well-established parts, it should surprise no one to learn that other languages have been built with similar designs. Of particular interest are C++ and Cyclone, the two languages closest to Rust.

7.1 CYCLONE

The Cyclone programming language is easily the closest relative to Rust. Cyclone was a project with similar goals to Rust: make a language that can do all the low-level stuff normally reserved for C and C++, but make it safe. However, while Rust was designed as a completely new language, Cyclone was designed to be a minimal addition on top of C. The ideal of Cyclone was that existing C code could be migrated to it relatively painlessly. This ultimately limited Cyclone's semantics, resulting in a system that was more awkward to use than Rust.

The Cyclone project was officially abandoned with its 1.0 release in 2006. Its official website links only to Rust for those interested in Cyclone's ideas [4]. Rust's official website in turn cites Cyclone as an influence for region analysis [8].

Early versions of Cyclone did not include affinity at all, as all types in C are always copyable. Cyclone also initially used region analysis primarily to manage dynamic pools of heap allocations, in much the same way that Tofte and Talpin's MLKit work did. In order to be sound, these early versions had to make C's `free` function a no-op, leaking the memory [20].

However the final release of Cyclone introduced several constructs that made it quite close to Rust 1.0 [23]. In particular, Cyclone introduced a *unique pointer* type which could be manually allocated and freed. In order to avoid use-after-frees, these pointers had to be affine. Internal references to a unique pointer could be taken, and were tracked in much the same way internal references are tracked in Rust. Unfortunately, no general mechanism was exposed for making arbitrary values or types affine, making this only usable for managing allocations. Since the project was officially abandoned after the release of unique pointers, we do not believe that affinity was explored in any further depth in Cyclone.

7.2 C++

C++ is the 8000-pound gorilla of systems programming in more ways than one. Not only is it one of the two major choices for projects that want low-level control, it's also incredibly complicated. Just like Cyclone, an initial desire to be a drop-in replacement for C resulted in significantly more complicated ownership semantics.

C++'s initial solution to the problem of pervasive copying in C was to introduce the ability to overload the behaviour of assignment and copying. In C if a value semantically owns an allocated buffer, copying that value will create two values with the the same buffer pointer, which isn't correct. In C++, that type would probably overload copying to create a new allocation for the copy. This does indeed solve the problem, but introduces several new problems.

The ability for every assignment and copy to execute arbitrary code makes it much more difficult to reason about correctness and performance of a piece of code. For instance, the statement 'x = y' could involve allocating and copying a massive buffer. In addition, this creates a serious exception-safety problem, as now *everything* can throw an exception.

For these reasons, Rust has specifically avoided overloading copying and assignment, defining both to just always be a bitwise copy. This is possible in Rust because values that own resources are affine, so the compiler will appropriately mark them as unusable, preventing the duplicated pointer problem.

To avoid the cost of expensive copies, C++11 introduced *move semantics*. Moving a value to a new location allows the old location to be cannibalized for the resources it owns, with the new location taking ownership of them. This is the same basic idea as affine typing in Rust, but with two major differences: the old location must still contain a valid instance, and arbitrary code can be executed to complete the move. Once again, these constraints leave C++'s solution significantly more difficult to reason about and use correctly.

C++'s unique_ptr type is a pointer to an owned value on the heap. When moved, the old location's value is set to 0, indicating that there is no allocation. Dereferencing such a pointer is Undefined Behaviour, but because the value is "valid", C++ will do nothing to prevent someone from doing this:

```cpp
#include <iostream>
#include <memory>

int main() {
    // x owns the allocation
    auto x = std::make_unique<int>(0);
    // now y owns the allocation
    auto y = std::move(x);
    // dereference null x; oops!
    std::cout << *x;
    std::cout << *y;
}
```

Even when passed the most pedantic warning flags, the latest version of clang happily compiles this program without any indication that something bad could happen. The equivalent Rust program fails to compile:

```
fn main() {
    let x = Box::new(0);
    let y = x;
    println!("{}", x);
    println!("{}", y);
}
```

```
<anon>:4:20: 4:21 error: use of moved value: ‘x‘ [E0382]
<anon>:4     println!("{}", x);
                            ^
```

It can also do less work, because x does not need to be overwritten in Rust. The fact that x has been moved out of can almost always be statically tracked.

It should be noted that C++'s solution does have one explicit advantage: it can encode types which have significant locations. For instance, one could construct a linked list with nodes stored on the stack in C++, and have the move constructor for a node update the pointers of its neighbors. In Rust this design would never be sound, because a value isn't ever informed when it's moved. However it's the Rust developer's opinion that this is a relatively minor loss compared to the other benefits of Rust's solution.

C++ has no solution to the problem of borrowed pointers. Any interface that exposes an untracked pointer or reference is fundamentally unsafe, because they can be made to dangle. If C++ code wishes to be safer, this necessarily implies copying or moving the data, which is often less efficient than passing a reference (especially because copies and moves invoke arbitrary code).

However Bjarne Stroustrup, the creator of C++, is currently working on a project to resolve this issue. The *C++ Core Guidelines* [2] are an attempt to extend C++ with some new semantics, and then identify a safe subset which can be compiler verified. This project was announced only a few months ago, so little analysis of the system exists.

Our understanding is that rather than using the well-established system of region analysis, it's based on a custom system that maintains lists of what points to what. For simple cases it seems to behave in much the same way as Rust's system, and requires similar annotations. However we have been unable to get in touch with the developers of the system to discuss the details, which are unclear from the publicly available details. Only time will tell what becomes of this experiment.

8 CONCLUSION

We have demonstrated how several classic safety and correctness issues can be reduced to an issue of trust between code and the data it shares. In particular,

TODO: still this.

9 APPENDIX: RUST SYNTAX PRIMER

Since this thesis is focused on Rust in particular, it will include a lot of Rust source code. Although our code is slightly unidiomatic in order to avoid overloading the reader with the finer details of Rust's syntax and semantics, this section exists for readers who are completely unfamiliar with Rust. A significant portion of Rust's syntax and semantics will be explained as it's needed throughout the thesis, but the following doesn't really fit in anywhere. If one is interested in learning Rust properly, *The Rust Programming Language* is the officially recommended book, as it is free and written by the Rust developers.

9.1 BASIC SYNTAX

First and foremost, Rust has the same basic control constructs and operators as every other C-style language. *if*, *else*, *while*, and *return* behave exactly as expected. Assignment is performed with =, comparison with ==, and so on. The only notable exception is that Rust does not provide the traditional C-style for-loop. Instead, Rust only provides the simpler for-each construct:

```
for x in iter {
    // do stuff with x
}
```

Variables in Rust are declared with a *let statement*. Variables are immutable by default, and can be made mutable with the *mut* keyword. The type of a variable is usually inferred, but can be explicitly specified with a colon.

```
let x = true;        // x is an immutable boolean
let mut y = false;   // y is a mutable boolean
let z: i32 = 0;      // z is an immutable 32-bit integer
```

Functions in Rust are written as follows:

```
// A function called ‘add‘ that takes two 32-bit integers
// and returns their sum (also a 32-bit integer).
fn add(x: i32, y: i32) -> i32 {
    return x + y;
}
```

There are two major composite data types in Rust, *structs* and *enums*. Structs are exactly like structs in C – a series of named and typed fields that are all stored inline. Fields can be accessed through the *dot operator*, as in most C-like languages.

```
// A struct called Foo
struct Foo {
    x: u32,         // x is a u32
    y: bool,        // y is a boolean
}

// Get the x field of the given Foo
fn get_x(data: Foo) -> u32 {
    return data.x;
}
```

Enums, on the other hand, are a bit different from C. Like C enums, Rust enums are a simple way to represent a type whose value can be one of a small set of *variants*. However Rust enums are *tagged unions*, which means each variant can have arbitrary data associated with it.

```
// An enum called Bar with three variants A, B, and C
// A contains a u32
// B contains a boolean
// C has no data associated with it
enum Bar {
    A(u32),
    B(bool),
    C,
}
```

In order to access the data in an enum's variant, one must pattern match against the enum to determine what state it is in. Matching and binding the data to a variable is performed as a single step to ensure that one does not access the data as if it were a different variant.

```
// An exhaustive match of all possible variants that the value
// ‘data‘ could have.
match data {
    A(count) => {
        return count;
    },
    B(is_one) => if is_one {
        return 1;
    } else {
        return 0;
    },
    C => {
        return -1;
    },
}

// A fallible match of only a single variant. The code inside
// an ‘if-let‘ is executed only if ‘data‘ matches the pattern.
if let A(count) = data {
    return count;
}
```

Rust uses & for the address-of operator, as well as the pointer type itself. ∗ is the dereference operator. The dot operator will automatically dereference pointers as necessary.

```
let x: i32  = 0;       // x is an i32
let y: &i32 = &x;      // y is a reference to an i32
let z: i32  = *x;      // z is an i32 (copied from x)

struct Foo { data: i32 }
let x: Foo = Foo { data: 0 };
let y: &Foo = &x;
let x: i32 = y.data;   // automatic dereferencing with ‘.‘
```

9.2 GENERICS AND IMPLEMENTATIONS

Functions can be associated with a particular type by using an *impl* block. This is just a convenience for grouping functionality, and does not imbue the code with any particular semantic. However it is necessary to implement *traits* which are Rust’s version of an interface.

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

// Associate the following code with a Foo
impl Foo {
    // A static function, in this case a constructor.
    // ‘new‘ has no special meaning in Rust, it’s just a convention.
    // Invoked as ‘Foo::new()‘
    fn new() -> Foo {
        return Foo { x: 0, y: 1 };
    }

    // A member function that takes a Foo by-reference
    // invoked as ‘some_foo.get_x()‘
    fn get_x(&self) -> i32 {
        return self.x;
    }
    // A member function that takes a Foo by-value
    // invoked as ‘some_foo.consume()‘
    fn consume(self) {
        let id = self.get_x();
        delete_the_database(id);
    }
}

// An interface for things that can be converted to integers.
trait ToInt {
    fn to_int(&self) -> i32;
}

// Implementing that interface for Foo
impl ToInt for Foo {
    // Trait implementations are invoked just like
    // any other member function
    fn to_int(&self) -> i32 {
        return self.x;
    }
}
```

impl, fn, struct, enum, and trait can all be generic over arbitrary types. Generics in Rust are implemented in a similar manner to C++ templates, but are type-checked before any usage like Java generics. In order to facilitate this, generic arguments can be specified to implement particular traits.

```
// ‘Compare‘ is generic over some type T, allowing
// implementors to be compared against multiple things.
trait Compare<T> {
    fn is_larger_than(&self, other: &T) -> bool;
}

// The ‘Generic‘ struct can be made for any type.
struct Generic<T> {
    data: T,
}

struct Concrete {
    data: i32,
}

// Implementing the Compare trait concretely
impl Compare<Concrete> for Concrete {
    fn is_larger_than(&self, other: &Concrete) -> bool {
        return self.data > other.data;
    }
}

// Because Compare is generic, we can implement it multiple
// times, as long as the generic arguments are different.
impl Compare<i32> for Concrete {
    fn is_larger_than(&self, other: &i32) -> bool {
        return self.data > other;
    }
}

// Implementing the Compare trait generically. We declare that
// this implementation applies only if T itself implements Compare
// with a ‘where‘ clause.
impl<T> Compare<Generic<T>> for Generic<T>
    where T: Compare<T>
{
    fn is_larger_than(&self, other: &Generic<T>) -> bool {
        return self.data.is_greater_than(&other.data);
    }
}

// A function generic over any two types which can be compared.
fn compare<R, L: Compare<R>>(left: &L, right: &R) -> bool {
    return left.is_larger_than(right);
}
```

Traits implementations may have specific types associated with them. For instance, if we wanted to make Compare incredibly generic, we could make the result of `is_larger_than` be defined by the implementor:

```
trait Compare<T> {
    // The type of Result must be given by the implementor
    type Result;

    // And must be used as the return type for is_larger_than
    fn is_larger_than(&self, other: &T) -> Self::Result;
}

impl<T> Compare<Generic<T>> for Generic<T>
    where T: Compare<T>
{
    // My result type is bool
    type Result = bool;
    fn is_larger_than(&self, other: &Generic<T>) -> bool {
        return self.data.is_greater_than(&other.data);
    }
}
```

Associated types are essentially the same as generic trait arguments, except that they cannot be declared independently. Because Compare is Generic over T, a type can declare that it can be compared to different things. Previously we used this to specify that a Concrete could be compared to another Concrete, or an i32. However associated types don't let us do this. The associated types must be uniquely determined by the generic arguments. If a trait has no generic arguments, but does have associated types, that means it can only be implemented once per type.

9.3 CLOSURES

Closures are anonymous functions that can refer to local variables that are defined outside of the function. Closures are declared with pipes, as follows:

```
fn main() {
    let mut x = 0;
    // A closure that takes an `increment`
    // and increases `x` by that amount.
    let mut func = |increment: i32| { x += increment; };

    func(1);    // x += 1;
    func(2);    // x += 2;
}
```

How values should be captured is generally inferred based on usage, but Rust will only do this for capturing things by-reference. To indicate that something should be captured by value, the move keyword must be used:

```
fn main() {
    let x = 0;

    // A closure that takes an `increment`,
    // increases `x` by that amount, and returns it.
    // Effectively takes a snapshot of what `x` was
    // when the closure was made.
    let func = move |increment: i32| {
        return x + increment;
    };
    let y = func(1);
    let z = func(2);

    // prints 1
    println!("{}", y);
    // prints 2, because each call uses a fresh copy of ‘x‘
    println!("{}", z);
}
```

BIBLIOGRAPHY

[1] Indexing github repository. https://github.com/bluss/indexing. Accessed: 2015-11-30.

[2] C++ core guidelines github repository. https://github.com/isocpp/CppCoreGuidelines. Accessed: 2015-11-30.

[3] Crossbeam github repository. https://github.com/aturon/crossbeam. Accessed: 2015-11-23.

[4] Cyclone homepage. https://cyclone.thelanguage.org/. Accessed: 2015-11-30.

[5] Openjdk arraylist source. http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/00cd9dc3c2b5/src/share/classes/java/util/ArrayList.java#l771. Accessed: 2015-11-23.

[6] petgraph github repository. https://github.com/bluss/petulant-avenger-graphlibrary. Accessed: 2015-11-30.

[7] lrs-lang github repository. https://github.com/lrs-lang/lib, . Accessed: 2015-11-30.

[8] Rust reference manual, appendix: Influences. http://doc.rust-lang.org/reference.html#appendix:-influences, . Accessed: 2015-11-30.

[9] Announcing rust 1.0. http://blog.rust-lang.org/2015/05/15/Rust-1.0.html, . Accessed: 2015-11-23.

[10] Making sense of rust. http://www.mpi-sws.org/~dreyer/talk.pdf, . Accessed: 2015-11-23.

[11] Servo homepage. http://servo.org/, . Accessed: 2015-11-23.

[12] Servo github repository. https://github.com/servo/servo, . Accessed: 2015-11-23.

[13] Learning rust with entirely too many linked lists. http://cglab.ca/~abeinges/blah/too-many-lists/book/. Accessed: 2015-11-30.

[14] Brian Anderson, Lars Bergstrom, David Herman, Josh Matthews, Keegan McAllister, Manish Goregaokar, Jack Moffitt, and Simon Sapin. Experience report: Developing the servo web browser engine using rust. *CoRR*, abs/1505.07383, 2015. URL http://arxiv.org/abs/1505.07383.

[15] David Aspinall and Jaroslav Ševčík. Formalising java's data race free guarantee. In *Theorem Proving in Higher Order Logics*, pages 22–37. Springer, 2007.

[16] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[17] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[18] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *ACM SIGPLAN Notices*, volume 42, pages 315–326. ACM, 2007.

[19] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, 2011.

[20] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *ACM Sigplan Notices*, volume 37, pages 282–293. ACM, 2002.

[21] Philip Munksgaard and Thomas Bracht Laumann Jespersen. *Practical Session Types in Rust*. PhD thesis, Master's thesis, Department of Computer Science, University of Copenhagen, 2015.

[22] Bruce Schneier. Heartbleed. *Schneier on Security*, 9, 2014. Accessed: 2015-11-28.

[23] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in cyclone. *Science of Computer Programming*, 62(2):122–144, 2006.

[24] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and computation*, 132(2):109–176, 1997.

[25] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Undefined behavior: what happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*, page 9. ACM, 2012.

[26] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *ACM SIGPLAN Notices*, 33(5):249–257, 1998.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of December 5, 2015 (`classicthesis` version 4.1).