Understanding Over Guesswork

Evolving How We Learn Systems with Lessons from Programming in the Large

Andrew Hobden and Yvonne Coady

Computer Science, University of Victoria

ABSTRACT

Some bugs are just that — a one off. A wayward moth that just happens to be innocently fluttering through the wrong relay at the wrong time. But some kinds of bugs aren't like that. Instead, they have risen to superstar popularity, plaguing veterans and newcomers alike. But what if these aren't bugs at all? What if they are actual deficiencies in safety and robustness offered by the C programming language as a consequence of the degree to which guesswork is introduced. Here we explore a more explicit approach to systems level programming supported by Rust, which we believe will better promote understanding of design intent, and eliminate some of the guesswork. We consider this in the context of an offering of a typical OS course, where students often first encounter these deficiencies, and in light of the classic bugs identified almost 15 years ago by Engler.

1. INTRODUCTION

Concurrency, parallelism, memory management, process scheduling, deadlocks, mutexes, system calls, filesystems, and architectural considerations are all commonly taught concepts in Operating Systems courses. These topics can be a struggle to understand, even for determined students, due to their complex, low-level characteristics.

Instructors may also find themselves struggling, as these assignments can be difficult to create, and at times nearly impossible to evaluate effectively. Instructors and their markers desire assignments which are simple enough to fit into a few files, demonstrate understanding of failure modes, can be tested effectively in an automated fashion, and show students the caveats of their attempts to solve the problem. In many cases, a trade-off is necessary. For example, building an interactive shell is a common, and much loved, assignment in which instructors must balance the number of features required with the time provided. Features such as pipes, background tasks, tab-completion, and environment variables are all desirable and interesting to implement, but contribute greatly to the complexity of the code, as well as the amount of time it takes to evaluate.

On top of the complexity, the ambiguity of language features means that year after year every set of new students hit the same old bugs—eventually. Engler et al. [21] identified a number of problem classes in their work in static analysis that offer students and professionals alike stumbling blocks, namely:

- Can routine `F` fail?
- Must `A` be paired with `B`?
- Does security check `Y` protect `X`?
- Can `A` be done after `B`?
- Does lock `L` protect `V`?

This short paper demonstrates precisely how Rust addresses these potential bugs in a clear, clean, safe and robust manner. After introducing Rust (Section 2), we discuss how Rust approaches and helps solve to these common bug categories. In Rust, routines with the potential for failure carry it explicitly in their function signature (Section 3), RAII is used to ensure allocations are followed by frees (Section 4), security checks can be required by the type system or through marker traits for 'tainted' data (Section 5), powerful move semantics eliminate use-after-free errors (Section 6), and locks inherently protect data, not code (Section 7). We also discuss the goals of "Safety" Rust (Section 8) state of tooling (Section 9), and the Rust community (Section 10).

2. INTRODUCING RUST

Rust [16] is a systems oriented ML-family language supported by Mozilla Research. It was originally conceived by Graydon Hoare and reached its first stable release on May 15, 2015 [3]. It is dual licensed Apache and MIT, fully open source, and governed through an extensive Request For Comment (RFC) process.

Rust offers a robust set of desirable features for systems code:

- Ahead-of-time compilation
- Zero-cost abstractions
- Move semantics
- Guaranteed memory safety
- Threads without data races
- Trait-based generics

- Pattern matching
- Type inference
- Minimal runtime (removable, Reference 8)
- No garbage collector or VM necessary
- Efficient C bindings
- Robust static analysis

It accomplishes these features through a number of novel techniques largely built off its type system and the borrow checker. The Rust community has been working to firmly position Rust as a powerful tool for programming in ultra-large, [6], embedded, and networking systems.

2.1 Rust Basics

To someone familiar with C/C++ the syntax of Rust will appear reasonably familiar. Rust differs in many ways though, believing in that it is better to be explicit and promote understanding of what is occurring, than to expect the programmer to maintain all of this information in their head and engage in guesswork.

This is a key motivating factor behind our proposed adoption of Rust in OS courses, we believe this quality does not do away with conciseness or elegance of code. Community members have developed bindings for well-known tools like Redis [12] and found the APIs for equivalent Rust and Python actions of relatively similar "feel", despite the benefits of Rust's type system providing an additional safety net [1].

Rust does, however, have significant semantic differences compared to C-like languages. For variable declaration, Rust has the `let` keyword which is *immutable by default*, mutability is opt-in via `let mut`. This opt-in mutability was found by the community to encourage better code. Instead of the programmer needing to remember to use `const` the compiler informs them of any variables they might have forgotten to make mutable or if it is unnecessarily mutable.

As well, function definitions differ from C-like languages. This change makes function definitions easier to comprehend when dealing with complex parameters, generics, and return values. Numerous reasoning for why C's declaration syntax is inadequate were well explained by Rob Pike [7].

```
fn example_simple()
fn example_params(x: u64, y: &u64, z: &mut u64)
fn example_returns(x: u64) -> u64
fn example_generic<U: Read>(reader: U) -> u64
fn example_generic_alt<U>(reader: U) -> u64
    where U: Read
```

2.2 A Strong Type System

While a dynamic type system is desirable in some areas, particularly in higher level code, things like implicit, possibly lossy data conversions can often be dangerous in system code. In our experience, many operating systems students also struggle with the mental concepts of pointers and their uses. This can lead to taking pointers as values and performing pointer arithmetic.

The programmer is not *prevented* from doing these things in Rust, it only ensures that it is actually the intended action. For many students though, attempting to cast pointers into a value is actually a mistake in their intention. Rust helps users with this by automatically dereferencing pointers when necessary, and providing stronger tools for common places where these mistakes crop up, like string indexing or dynamic array access.

Types can be created easily, and there are three basic compound data structures, `struct`, `enum`, and tuples. `struct`s and tuples are similar to other languages. Rust's `enum`s are able to represent variants with encapsulated values, generics, and even `struct`s!

```
// Structure with generic
struct One<T> {
    foo: usize,
    bar: T
}
// 2-tuple
struct Two(usize, usize);
// Enum
enum Three {
    // Plain.
    Foo,
    // Variant with Tuple.
    Bar(usize),
    // Variant with Struct.
    Baz { x: u64, y: u64, z: u64, },
}
```

2.3 We Don't Need A `null`

Cited by its creator [9] as a 'billion-dollar mistake' `null` is one of the most dangerous thorns in a programmers toolbox. What's more is that these errors happen at *runtime* and may take down live systems.

In languages like C, C++, and Java a tremendous amount of research and development time has gone into developing products like Coverity [4] and PVS-Studio [11] to help discover possible null pointer inconsistencies. Engler et al suggest heuristic methods to determine the 'null state' of a variable throughout the control flow of a program. What if programmers could just stop worrying about `null` all together?

Many functional languages like Haskell and F# have the concept of an `Option`, a concept that Rust shares. Instead of needing to be aware of and check for `null` at every occurrence, the language semantics require the programmer to explicitly decide on the control flow for all values.

```
// Create a `Some(T)` and a None.
let maybe_foo = Some(0);
let not_foo = None;
// Unwrapping.
let foo = maybe_foo.unwrap();
let default = not_foo.unwrap_or(1);
let matched = match maybe_foo {
    Some(x) => x,
    None => -1,
};
// Mapping
let mapped = maybe_foo.map(|x| x as f64);
```

3. RESULTS AND `TRY!()`

When working with traditional languages such as C and C++ it can often be difficult to answer the question "Can this function fail?" Checked exceptions can help, but often APIs are inconsistent, and checks for failure can be forgotten [21]. Some static analysis techniques can be used to determine possible missed failure checks, such has detecting invocations that do check for error. Having failure information included in the function's signature and requiring it to be explicitly checked may be a more robust solution over heuristics though.

The `Result<T, E>` enum exists as either `Ok(T)` or `Err(E)` and conveys the result of something which may fail with an error. Using Rust's `match` expression the user can act on various error conditions or success.

```
use std::io;
use std::error::Error;
// Create an error. (Normally raised from lib)
let error = io::Error::new(io::ErrorKind::Other,
    "I'm an example error!");
// The two result variants. Type notations usually
// not necessary except in small examples.
let success: Result<_, io::Error> = Ok("Success!");
let failure: Result<&str, _> = Err(error);
// Return either the value or the error description.
let val_or_desc = match success {
    Ok(val) => val,
    Err(ref e)  => e.description(),
};
```

It is a compiler warning to perform an action such as `file.read_to_string(buf)` which returns a `Result<usize, Error>` and to not handle the error in some way. In Rust is it idiomatic for any recoverable error to be passed up the call stack to where it can be sensibly handled. While approaching this idea newcomers typically struggle with the fact that an `io::Error` and a `Utf8Error` are different types and cannot be returned in the same `Result<T,E>`, since the `E` value would differ and violate Rust's strong typing. This is typically solved by creating a new `Error` which is an enumeration over the possible underlaying errors as well as any the programmer may wish to include themselves. Then there are the `Into<T>` and `From<T>` traits which can be implemented to provide seamless interaction.

```
pub enum MyError {
    Io(io::Error),
    Utf8(Utf8Error)
}
impl From<io::Error> for MyError {
    fn from(err: io::Error) -> Error {
        Error::Io(err)
    }
}
// ...
```

When working with functions which may return a `Result<T, E>` it is common to use the `try!()` macro. This macro expands to either unwrap the `T` value inside and assign it, or return the error up the call stack. This helps reduce visual 'noise' and assist in composition.

```
fn open_and_read() -> Result<String, MyError> {
    let mut f = try!(File::open("foo.txt"));
    let mut s = String::new();
    let num_read = try!(f.read_to_string(&mut s));
    Ok(s)
}
```

Error handling in Rust is explicit, composable, and sane. There are no exceptions, nulls, 'magic numbers' (like -1) or anything that may prevent the programmer from handling the error as *they* choose to, even if that is to simply `.unwrap()` it and fail. It's worth noting that even `.unwrap()`ing does not actually crash the program as normally it unwinds the stack, isolating failure to a single thread and preventing inconsistent state.

4. BORROW AND MOVE: FORGET `FREE()`

In Rust there is the notion of moving, copying, and referencing. In some ways Rust's memory model is similar to C/C++'s. It features a powerful pointer system that allows programmers to make fine-grain, informed decisions about how values are stored, passed, and represented. Like C++, Rust makes use of a concept called Resource Acquisition is Instantiation (RAII). Rust goes a step further, introducing the distinction between *immutably borrowing* (`&`), *mutably borrowing* (`&mut`), *copying* (`Copy` trait), and *moving* values. At any given time there may be any number of *immutable borrows*, meanwhile there may only be one *mutable borrow*, and a value may not be used in the function once it has been *moved* out.

This makes it simple for a programmer to observe a function signature and determine which values the function may mutate or consume, and which it may return. Using this information the compiler is able to determine the lifetime constraints of almost any value without additional notations. In (rare, complex) cases where it does require additional information the programmer can annotate lifetimes just as they would generic type parameters.

```
fn main() {
    // An owned, growable,
    // non-copyable string.
    let foo = String::from("foo");

    // Introduce a new scope.
    {
        // Reference bar is created.
        let bar = &foo;
        // Error, bar is immutable.
        bar.push('c');
    } // bar is destroyed.

    // Error, bar does not exist.
    let baz = bar;
    // Works, reference mutable.
    let rad = &mut foo;
    rad.push('c');
} // foo is destroyed.
```

This behavior is very similar to C++'s RAII facilities and ensures all values are safely destructed in a consistent, predictable way as soon as they are no longer needed. The programmer does not need to worry about making sure each of their `malloc()` calls have a corresponding `free()` or rely on an outside tool [21] to discover such errors. The borrow checker is also able to determine when a value has been *moved* into a function call and should not be further used in the caller, eliminating another possible class of errors.

5. TRAITS: ZERO-COST ABSTRACTIONS

Rust does not use a class based or inheritance based system. Data is stored in `struct`s, primitives, or `enum`s which implement a set of traits that define how it interacts and which functions are available to it. For example, the `File` is a `struct` which implements `Read` and `Write` among other traits. Other structures like `TcpStream` and `UdpSocket` also implement the same `Read` and `Write` trait. Traits are zero-cost abstractions that act to encourage common interfaces and capabilities between like-structures [2].

```
struct Thing {
    barred: bool,
}
trait Foo {
    // Implementor must define.
    fn bar(&mut self);
    // Default definition.
    fn do_bar(&mut self) { self.bar() }
}
impl Foo for Thing {
    fn bar(&mut self) {
        self.barred = true;
    }
}
```

Traits fit easily together, are widespread in their implementation, and allow for common interfaces between modules to permit better adaptability. Traits can also be used as 'markers' in design patterns like state machines to provide additional compile time verification of correctness.

6. STATIC ANALYSIS AT THE CORE

Static analysis tools, like `splint` for C [17] are an invaluable tool for Operating Systems programming, particularly when working on large codebases with multiple programmers.

Rust's type system and region based memory, based off the ideas of Cyclone [14], are particularly well suited to static analysis. Indeed, `rustc` itself performs a tremendous amount of static analysis without the help of external tools. The type system carries all the information necessary for the compiler to understand all possible control flows of the program, all possible (recoverable) errors which arise, and the lifetimes of each region of memory.

Of particular interest is `rustc`'s "Borrow Checker" which analyzes and understands the pointer system and is able to verify data safety, even across multiple threads. The borrow checker is an area of active research [10] and has thus far proven itself sound.

As a result of the static analysis done by `rustc` it is able to infer information about (but is not limited to):

- Unused results, variables and functions.
- Unreachable code.
- Unsafe pointer sharing (multiple mutable pointers.)
- Incorrect type matching and lossy casts.
- Use-after-free errors.
- Unclear lifetimes (asking for either clarity or refactoring.)

7. THREADS THAT DON'T BITE

Threading is perhaps one of the most powerful and robust features of Rust. The characteristics detailed above culminate in a sort of *tour de force* when used bravely in a threaded context.

Harnessing the power of ownership semantics, the type system, the standard library's threading modules there are a number of tools [5]:

Channels provide a way to transfer messages (and ownership) between threads without fear of there being later (unsafe) access to the data by other threads. The vanilla channel provided by the standard library is a Multiple-Producer, Single-Consumer channel.

```
use std::sync::mpsc::{channel, Sender, Receiver};
let (send, recieve) = channel();
```

Locks can encapsulate data such that it can only be accessed if the lock is held. In Rust, you **don't lock code, you lock data**, and it is safer because of it. Locks are typically represented by `Mutex`s and shared between threads with an Atomically Reference Counted structure (`Arc`). It should be noted that this design of locking data prevents a lock from being required and never given up, a common mistake. [21]

```
use std::sync::{Arc, Mutex};
let data = Arc::new(Mutex::new(0));
```

Traits like `Sync` and `Send` are implemented on types and symbolize if it can be *sent* or *shared* between threads safely. These traits are not just documentation, they are intrinsic to the language.

```
// Safe to share between threads.
use std::marker::Sync;
// Safe to transfer between threads.
use std::marker::Send;
```

Other, more fearless forms of concurrency such as **sharing stack frames** is even encouraged by these models. This is done via a scoped thread model.

```rust
fn main() {
    let items = vec![1, 2, 3];
    let mut guards = vec![];
    for item in items {
        let guard = thread::scoped(move || {
            print!("{}", item);
        });
        guards.push(guard);
    }
} // `guards` destroyed here, implicitly joining
```

8. SAFETY AS A FIRST-CLASS GOAL

The concept of "Safety" in code is often poorly defined. In general, safety can be broken down into three categories:

- **Type Safety** is the ability of a language to prevent or discourage type errors, such as treating a `float` like an `int`. Rust and languages like Haskell excel here as their type systems are strong, explicit (but often inferred), and do not include the notion of a `null` that can go anywhere indiscriminately.
- **Memory Safety** is the ability of a language to reduce or eliminate the possibility of mistakes like writing a 64 bit value into a 32 bit space (overwriting unintended data), or multi-threaded mutable access to the same memory. Rust's borrow checker effectively eliminates data races in safe code and strong type safety prevents unintended clobbering.
- **Thread Safety** is the ability of a language to prevent inter-thread race conditions, such as one thread exiting when another thread is waiting on data from it. Rust provides some robust tools for managing thread pools integrated into its type system, however some mistakes are still possible if the programmer works hard enough to accomplish them.

Rust advertises both type safety and data safety, accomplishing both very effectively. There is still research and development to be done before it can truthfully bill itself as thread-safe, but this type of safety is perhaps the most elusive.

9. TOOLING

One significant advantage of using Rust over its alternatives is its robust, opinionated set of tooling. The Rust standard distribution includes `rustc` (the compiler), `cargo` (a package manager and build tool), and `rustdoc` (a documentation generator). Currently there is work being done on a `rustfmt` which would function the same as Go's venerable `gofmt`.

Package management via `cargo` is a feature Rust has inherited from several other modern languages. All package dependencies, build options, and tasks are defined in a `Cargo.toml` file. Dependencies are checked and (if necessary) pulled on `cargo build`, `test`, or `doc`.

Rust supports both *unit tests* and *integration tests* by default. Unit tests may appear wherever is appropriate in the code and are annotated by `#[test]`, it is common for designers to include a `test` module in their code. Integration tests are written in the `tests/` directory and allow a package to be tested as a depended upon library. Testing is done by simply invoking `cargo test` in the project directory. These features blow away barriers which programmers might face in other languages that would prevent them from bothering to test. Additionally, it makes marking Rust based projects very easy, all an instructor needs to do is provide (or replace) the `tests/` directory with an appropriate suite.

```rust
#[test]
fn test_passes() {
    assert_eq!(true, true);
}
#[test]
#[should_panic]
fn test_fails() {
    assert!(true == false);
}
```

Having a standardized, high quality documentation format is invaluable for programmers, and Rust facilitates this. Documentation comments are can be placed anywhere in the code using `///` for function level documentation or `//!` for module level documentation. Documentation is in a common markdown format, code samples included in the documentation are automatically processed as unit tests. Generating documentation is done by `cargo doc`, which generates HTML and manpage documentation. Many Rust projects even go so far as to automate the unit testing and documentation generation step and hook it into their git commits [15].

10. COMMUNITY

One of the biggest dangers in choosing a language that "Is not C" to teach operating systems in is that it can be very difficult for students to get help.

Mozilla's IRC network hosts the popular #rust channel which regularly has over 800 members at any given time. `crates.io` hosts over 2300 packages. The language reached 1.0 on May 15, 2015 [3] and has been in development since 2006. The community is active and friendly with a variety special interest groups.

Best of all, there is active operating system development in Rust. There is a project to develop `coreutils` [18], a kernel [19], operating systems [13], and embedded system platforms [20]. At the time of writing, these projects are young enough that students could even contribute components upstream.

11. FUTURE WORK

There is a considerable amount of research remaining regarding Rust's uses in systems code and programming in the large in general. We seek to foster knowledge of the language at the University of Victoria and are working on developing distributed consensus algorithms like Raft and next generation initialization systems in the spirit of OpenRC.

References

[1]A Fresh Look at Rust: *http://lucumr.pocoo.org/2014/10/1/a-fresh-look-at-rust/#designing-apis*. Accessed: 2015-06-30.

[2]Abstraction without overhead: traits in Rust: *http://blog.rust-lang.org/2015/05/11/traits.html*. Accessed: 2015-06-30.

[3]Announcing Rust 1.0: *http://blog.rust-lang.org/2015/05/15/Rust-1.0.html*. Accessed: 2015-06-30.

[4]Coverity: *https://www.coverity.com/*. Accessed: 2015-06-30.

[5]Fearless Concurrency: *http://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html*. Accessed: 2015-06-30.

[6]Feiler, P. et al. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon University.

[7]Go Blog: Function Declaration: *https://blog.golang.org/gos-declaration-syntax*. Accessed: 2015-06-30.

[8]libcore: *http://doc.rust-lang.org/book/no-stdlib.html*. Accessed: 2015-06-30.

[9]Null References: The Billion Dollar Mistake: *http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare*. Accessed: 2015-06-30.

[10]Patina: A Formalization of the Rust Programming Language: *ftp://ftp.cs.washington.edu/tr/2015/03/UW-CSE-15-03-02.pdf*. Accessed: 2015-06-30.

[11]PVS-Studio: *http://www.viva64.com/en/pvs-studio/*. Accessed: 2015-06-30.

[12]Redis: *http://redis.io/*. Accessed: 2015-06-30.

[13]Reenix: Implementing a Unix-Like Operating System in Rust: *https://scialex.github.io/reenix.pdf*. Accessed: 2015-06-30.

[14]Region-Based Memory Management in Cyclone: *http://209.68.42.137/ucsd-pages/Courses/cse227.w03/handouts/cyclone-regions.pdf*. Accessed: 2015-06-30.

[15]Rust, Travis, and Github Pages: *http://hoverbear.org/2015/03/06/rust-travis-github-pages/*. Accessed: 2015-06-30.

[16]Rust: *http://rust-lang.org/*. Accessed: 2015-06-30.

[17]Splint: *http://splint.org/*. Accessed: 2015-06-30.

[18]uutils/coreutils: *https://github.com/uutils/coreutils*. Accessed: 2015-06-30.

[19]Writing an OS in Rust in Tiny Steps: *http://jvns.ca/blog/2014/03/12/the-rust-os-story/*. Accessed: 2015-06-30.

[20]zinc.rs: *http://zinc.rs/*. Accessed: 2015-06-30.

[21]Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. *Eighteenth ACM symposium on Operating systems principles (SOSP '01)* (ACM, New York, NY, USA), 57–72.